Posibilidades del motor 3D de la Nintendo DS

Proyecto Fin de Carrera

Ingeniería Técnica de Telecomunicaciones
Especialidad Imagen y Sonido

Escuela Universitaria de Ingeniería Técnica de Telecomunicación
Universidad Politécnica de Madrid

Pablo Santiago Liceras

Tutor:
Enrique Rendón Angulo

9 de septiembre de 2013

Proyecto Fin de Carrera

Tema: Gráficos por ordenador

Título: Posibilidades del motor 3D de la Nintendo DS

Autor: Pablo Santiago Liceras

Titulación: Sonido e imagen

Tutor: Enrique Rendón Angulo

Departamento: DIAC

Miembros del Tribunal Calificador:

Presidente: Magdalena González Martín

Vocal: Enrique Rendón Angulo

Vocal Secretario: Alfonso Martín Marcos

Fecha de Lectura: Septiembre 2013

Resumen

La consola portátil Nintendo DS es una plataforma de desarrollo muy presente entre la comunidad de desarrolladores independientes, con una extensa y nutrida escena *homebrew*. Si bien las capacidades 2D de la consola están muy aprovechadas, dado que la mayor parte de los esfuerzos de los creadores amateur están enfocados en este aspecto, el motor 3D de ésta (el que se encarga de representar en pantalla modelos tridimensionales) no lo está de igual manera.

Por lo tanto, en este proyecto se tiene en vista determinar las capacidades gráficas de la Nintendo DS. Para ello se ha realizado una biblioteca de funciones en C que permite aprovechar las posibilidades que ofrece la consola en el terreno 3D y que sirve como herramienta para la comunidad *homebrew* para crear aplicaciones 3D de forma sencilla, dado que se ha diseñado como un sistema modular y accesible.

En cuanto al proceso de renderizado se han sacado varias conclusiones. En primer lugar se ha determinado la posibilidad de asignar varias componentes de color a un mismo vértice (color material reactivo a la iluminación, color por vértice directo y color de textura), tanto de forma independiente como simultáneamente, pudiéndose utilizar para aplicar diversos efectos al modelo, como iluminación pre-calculada o simulación de una textura mediante color por vértice, ahorrando en memoria de video. Por otro lado se ha implementado un sistema de renderizado multi-capa, que permite realizar varias pasadas de render, pudiendo, de esta forma, aplicar al modelo una segunda textura mezclada con la principal o realizar un efecto de reflexión esférica.

Uno de los principales avances de esta herramienta con respecto a otras existentes se encuentra en el apartado de animación. El renderizador desarrollado permite por un lado animación por transformación, consistente en la animación de mallas o grupos de vértices del modelo mediante el movimiento de una articulación asociada que determina su posición y rotación en cada frame de animación. Por otro lado se ha implementado un sistema de animación por muestreo de vértices mediante el cual se determina la posición de éstos en cada instante de la animación, generando frame a frame las poses que componen el movimiento (siendo este último método necesario cuando no se puede animar una malla por transformación).

Un mismo modelo puede contener diferentes esqueletos, animados independientemente entre sí, y cada uno de ellos tener definidas varias costumbres de animación que correspondan a movimientos contextuales diferentes (andar, correr, saltar, etc).

Además, el sistema permite extraer cualquier articulación para asociar su transformación a un objeto estático externo y que éste siga el movimiento de la animación, pudiendo así, por ejemplo, equipar un objeto en la mano de un personaje.

Finalmente se han implementado varios efectos útiles en la creación de escenas tridimensionales, como el *billboarding* (tanto esférico como cilíndrico), que restringe la rotación de un modelo para que éste siempre mire a cámara y así poder emular la apariencia de un objeto tridimensional mediante una imagen plana, ahorrando geometría, o emplearlo para realizar efectos de partículas. Por otra parte se ha implementado un sistema de animación de texturas por subimágenes que permite generar efectos de movimiento mediante imágenes, sin necesidad de transformar geometría.

Abstract

The Nintendo DS portable console has received great interest within the independent developers' community, with a huge homebrew scene. The 2D capabilities of this console are well known and used since most efforts of the amateur creators has been focused on this point. However its 3D engine (which handles with the representation of three-dimensional models) is not equally used.

Therefore, in this project the main objective is to assess the Nintendo DS graphic capabilities. For this purpose, a library of functions in C programming language has been coded. This library allows the programmer to take advantage of the possibilities that the 3D area brings. This way the library can be used by the homebrew community as a tool to create 3D applications in an easy way, since it has been designed as a modular and accessible system.

Regarding the render process, some conclusions have been drawn. First, it is possible to assign several colour components to the same vertex (material colour, reactive to the illumination, colour per vertex and texture colour), independently and simultaneously. This feature can be useful to apply certain effects on the model, such as pre-calculated illumination or the simulation of a texture using colour per vertex, providing video memory saving. Moreover, a multi-layer render system has been implemented. This system allows the programmer to issue several render passes on the same model. This new feature brings the possibility to apply to the model a second texture blended with the main one or simulate a spherical reflection effect.

One of the main advances of this tool over existing ones consists of its animation system. The developed renderer includes, on the one hand, transform animation, which consists on animating a mesh or groups of vertices of the model by the movement of an associated joint. This joint determines position and rotation of the mesh at each frame of the animation. On the other hand, this tool also implements an animation system by vertex sampling, where the position of vertices is determined at every instant of the animation, generating the poses that build up the movement (the latter method is mandatory when a mesh cannot be animated by transform).

A model can contain multiple skeletons, animated independently, each of them being defined with several animation customs, corresponding to different contextual movements (walk, run, jump, etc).

Besides, the system allows extraction of information from any joint in order to associate its transform to a static external object, which will follow the movement of the animation. This way, any object could be equipped, for example, on the hand of a character.

Finally, some useful effects for the creation of three-dimensional scenes have been implemented. These effects include billboarding (both spherical and cylindrical), which constraints the rotation of a model so it always looks on the camera's direction. This feature can provide the ability to emulate the appearance of a three-dimensional model through a flat image (saving geometry). It can also be helpful in the implementation of particle effects. Moreover, a texture animation system using sub-images has also been implemented. This system allows the generation of movement by using images as textures, without having to transform geometry.

Dedico el presente trabajo a mis padres, Mara y José María. A mi madre por la ayuda y el soporte que me ha brindado, tanto a la hora de guiarme, como simplemente de escucharme hasta encontrar la solución a un problema. A mi padre por sus consejos, que tratan siempre de transmitirme su experiencia. Y a ambos por la educación que me han ofrecido y los valores que me han inculcado, enormes e impagables.

A mi hermano Carlos, que siempre ha sido un gran apoyo técnico y fuente de crecimiento científico y personal.

A mi abuela y mis tíos, que hacen más fácil y divertida la existencia.

A mis suegros, Jesús y Manoli, por acogerme y darme apoyo y cariño.

A Enrique Rendón, que me ha hecho descubrir mi pasión y el camino profesional que quiero tomar, y por su dedicación a este proyecto.

Y a Andrea. Porque tu apoyo, determinación y paciencia, a lo largo de este proyecto, en la carrera, en el colegio, y en el resto de momentos de mi vida, han sido determinantes para llegar a ser la persona y el profesional que ahora soy. Te quiero.

Tabla de contenidos

Índice de figuras

Capítulo 1

Introducción

Este documento contiene la memoria del presente Proyecto Fin de Carrera, "Posibilidades del motor 3D de la Nintendo DS", englobado en el campo de los gráficos por ordenador y perteneciente al departamento de Ingeniería Audiovisual y Comunicaciones de la Universidad Politécnica de Madrid.

1.1. Motivación

La informática gráfica es un ámbito de investigación y desarrollo en auge, empleado en multitud de sectores, como la formación mediante simuladores, la medicina, la educación o el ocio, ya sea cinematográfico o de la industria de los videojuegos. Por ello, se emplean gran cantidad de recursos en el avance de esta materia, tanto a nivel de tecnología como de aplicación.

Por otro lado, la facilidad de comunicación e intercambio que aportan las nuevas tecnologías de la información ha acercado este sector al gran público, haciéndose más fácil desarrollar aplicaciones y herramientas relacionadas con esta materia. Así, se forman comunidades de desarrollo abiertas que comparten tanto recursos como información relativas a un tema en concreto.

La videoconsola Nintendo DS es un claro ejemplo de esto, ya que cuenta con una gran escena de desarrollo abierto e independiente (*homebrew*), sustentada por un buen número de aficionados que crean y comparten herramientas y aplicaciones (especial mención a la interfaz de programación Libnds), tanto para desarrolladores como para usuarios finales. Por esta razón es una gran plataforma de desarrollo abierto.

Sin embargo, si bien se aprovechan las capacidades 2D de la consola, tanto en juegos como en aplicaciones, no es el caso de los gráficos 3D. En este caso hay muchos menos desarrollos, no encontrando demasiadas herramientas dedicadas a este campo, e incluso la información disponible para el aprendizaje es escasa y difusa.

Adicionalmente, las aplicaciones existentes en la escena *homebrew* que hace uso de las capacidades 3D de la Nintendo DS no aprovechan las posibilidades que ésta ofrece, limitándose al simple pintado poligonal con texturizado. Además, la animación es en la mayoría de casos inexistente, siendo éste un punto abandonado del desarrollo 3D *homebrew* en la Nintendo DS.

Por lo tanto, estas son las razones que han impulsado la realización de este proyecto: estudiar las capacidades del motor 3D de la Nintendo DS con el fin de desarrollar una biblioteca o herramienta de programación, utilizable por la comunidad *homebrew*, que las emplee. De esta forma se creará un sistema sólido de pintado de modelos 3D y animación, que además facilite el desarrollo en este campo.

1.2. Objetivos

Este proyecto consta de dos objetivos primordiales, los cuales se abordarán de forma simultánea, ya que ambos están fuertemente ligados.

El primero es estudiar los aspectos clave de la Nintendo DS a nivel de hardware, recurriendo a las especificaciones del procesador gráfico de la consola, proceso esencial para el aprovechamiento de las capacidades de éste en la herramienta de pintado tridimensional. Además se determinarán las herramientas necesarias que conforman el entorno de desarrollo *homebrew* de la Nintendo DS con el cual crear aplicaciones para ésta.

El segundo objetivo consiste en el desarrollo de una biblioteca de funciones de renderizado (pintado de modelos 3D) accesible y modular, en el lenguaje de programación C. Como ya se ha comentado, esta herramienta tiene como objetivos aprovechar las características gráficas de la Nintendo DS y facilitar el desarrollo de aplicaciones 3D a los desarrolladores *homebrew*.

Los objetivos en cuanto a las funcionalidades clave que se requieren implementar en esta herramienta son:

- **pintado de geometría** (vértices con normales)
- **texturizado** (esto implica la gestión de recursos, en este caso texturas)
- **color por vértice**
- **animación** (se estudiarán las distintas posibilidades)
- **renderizado multi-pasada** (capas adicionales de segunda textura y reflexión esférica)
- ***billboarding***
- **animación de texturas** por subimágenes

Esta biblioteca, basada en la parte gráfica de Libnds de DevkitPro, se implementará mediante funciones de acceso sencillas de emplear, que conformen una capa de abstracción superior al proceso de pintado 3D, haciendo de éste una operación transparente al usuario.

Además, a lo largo de este desarrollo, y mediante la experimentación, se comprobarán las capacidades gráficas de la Nintendo DS.

1.3. Sumario

Capítulo 2: introducción a los gráficos por ordenador. En este capítulo se definirán los conceptos básicos de este campo, enlazando con las etapas de producción que intervienen. Además, se hará una breve explicación de la cadena de procesamiento del renderizado de

modelos 3D (*pipeline* gráfico).

Capítulo 3: características y especificaciones técnicas de la Nintendo DS. Se detallarán los registros que conforman el motor 3D de la consola, cuyo manejo es esencial para gestionar los estados del hardware y así aprovechar sus capacidades.

Capítulo 4: generalidades de la parte gráfica de la interfaz de licencia libre Libnds, empleada como capa de soporte a la librería de renderizado desarrollada, especialmente el módulo de gráficos 3D.

Capítulo 5: trabajo de investigación y toma de decisiones previo al desarrollo. Esto deriva en el diseño e implementación (externos al proyecto y paralelos al desarrollo de la librería msNDS) de un formato de modelo específico para la Nintendo DS, Extended Display List (EDL), del cual se detallará su especificación. Adicionalmente se determinarán las herramientas externas necesarias para exportar y cargar imágenes en la consola, en los formatos que ésta admite.

Capítulo 6: especificación y documentación de la biblioteca de funciones de renderizado msNDS desarrollada en este proyecto. Se detallarán el entorno de usuario, conjuntamente con un pequeño manual de utilización, y el funcionamiento interno del sistema de renderizado implementado, finalizando con un ejemplo de uso.

Capítulo 7: recopilación de las capacidades gráficas de la Nintendo DS, unas propias del hardware y otras explotadas en la herramienta de pintado 3D msNDS. Se comentarán las limitaciones del procesador gráfico de la consola, así como las dificultades encontradas en cuanto a la implementación de ciertas funcionalidades.

Capítulo 8: conclusiones y líneas de trabajo futuro extraídas del desarrollo de la librería msNDS y del trabajo de recopilación efectuado a lo largo del proyecto.

Capítulo 2

Gráficos por ordenador

Para entender el desarrollo de este proyecto es esencial conocer aspectos clave del proceso de creación de gráficos por ordenador (*Computer Graphics*, en algunos textos aparece como CG). Se entiende como gráficos por ordenador al proceso de construcción, modificación y posterior manipulación de modelos geométricos e imágenes de objetos mediante el uso de un ordenador. A esta definición, y según la aplicación, se han de sumar la animación de dichos modelos, para crear mundos con vida, y la interacción con el usuario, muy presente en gran cantidad de aplicaciones de gráficos por ordenador [20].

Los inicios de los gráficos por ordenador se remontan a mediados del siglo XX [6], empleados esencialmente en aplicaciones militares (simuladores) o de ingeniería (modelado de automóviles). Sobre los años 70 aparecieron los primeros videojuegos primitivos, siendo su auténtico auge en la década de los 80. No es hasta los años 90 cuando empiezan a crearse los primeros cortos de animación, así como el primer largometraje animado (Toy Story, de Pixar, 1995).

En la actualidad, los gráficos por ordenador se emplean en multitud de aplicaciones, que van desde el ocio (videojuegos, películas de animación y efectos especiales), a la formación (simuladores) o la ciencia, la medicina y la ingeniería.

Se estudiará su desarrollo, de una forma muy general, ya que es un tema tremendamente extenso y en evolución constante, y podría ser materia de un estudio independiente. Por esta razón, lo que se explica en este texto no es la única vía de acción, ya que en cada etapa existen múltiples métodos para realizar un mismo proceso. Veremos por lo tanto el proceso desde el modelado y la animación, pasando por el denominado *pipeline* de renderizado de gráficos (siendo este campo el que realmente nos ocupa en este proyecto) y resultando en la interacción con el usuario final a través del interfaz de usuario. Además, veremos cómo evolucionan y varían las técnicas en ciertos puntos clave del proceso, según el entorno de desarrollo.

2.1. Modelado

El modelado es la etapa del proceso en la cual se genera un objeto geométrico por medios informáticos. Esta fase del desarrollo requiere unas herramientas de software muy específicas, concebidas para tal efecto y que difieren entre sí dependiendo de la aplicación final. Por ejemplo, para el tema que nos ocupa, que es la aplicación para videojuegos,

existen varios programas de modelado relativamente extendidos.

2.1.1. Herramientas de modelado

3D Studio Max [2] y Maya [3] de Autodesk (existen otros como SoftImage, Modo, Lightwave 3D...) son los más avanzados y los que se emplean de forma profesional, tanto para el desarrollo de videojuegos como para la animación (aunque por tradición se suele usar más 3D Studio Max para videojuegos y Maya para animación). Ambos programas tienen muchas similitudes, diferenciándose básicamente en su concepción: Maya originariamente pertenecía a Silicon Graphics Inc., pero fue comprado por Autodesk recientemente. Estas herramientas son tremendamente potentes y precisas, por lo que son realmente caras si se desean todas las características de la aplicación (existen versiones gratuitas pare estudiantes, pero si se utilizan para uso comercial hay que pagar licencia).

Como alternativa accesible y económica está Blender [4], una herramienta tan potente como las anteriores, que se beneficia de ser *open source*, por lo que está abierta a todo el que quiera modelar con precisión y detalle, además de que está en constante evolución. Es una opción muy interesante si se tiene en vista sacar un producto al mercado con un presupuesto limitado.

Para cerrar esta lista, y aunque existe más software dedicado al modelado, mencionaremos al programa que nos ocupa en este proyecto, y con el cual se ha desarrollado esta etapa de modelado: MilkShape 3D [14]. Es un programa muy básico, con opciones limitadas, pero muy fácil de usar y de aprender, suficiente para crear los modelos sencillos que requiere la Nintendo DS.

2.1.2. Proceso de modelado

En la etapa de modelado es donde se crean los objetos geométricos tridimensionales que más tarde se pintarán en el dispositivo de visualización. Estos modelos siempre tienen la misma información referente a su geometría y apariencia: vértices, normales, coordenadas de textura y apariencia de sus materiales. Si además el modelo es animado, éste tendrá información sobre sus huesos y las claves de animación, cómo se explicará más adelante.

Un modelo tridimensional se crea generando lo que se denominan mallas. Una malla es un conjunto de puntos en el espacio unidos entre sí. Estos puntos son vértices. Éstos se definen por su posición en el espacio o coordenadas. Así, un vértice tiene que tener como información básica cada una de las coordenadas (x, y, z) que definen su posición en el espacio tridimensional.

Los vértices que componen la malla se unen entre sí formando polígonos. Éstos polígonos son los que definirán una aproximación lineal a la superficie del modelo. Por lo tanto, cuantos más polígonos, más realista será el objeto modelado (menos "cuadrado"), pero más memoria y capacidad de procesamiento será requerida para procesar la información de tal cantidad de vértices y polígonos.

Existen varias técnicas de modelado. La más básica es la denominada modelado poligonal. En esta técnica, la forma básica de modelado son los polígonos de cuatro lados o *quads*, que el programa de modelado, al exportar el modelo a un determinado formato,

se encargará de agruparlos en grupos de tres, formando triángulos. Esto es así ya que tradicionalmente los renderizadores manejaban la información de esta manera, y si no le llegaba un triángulo podría ser causa de errores.

Sin embargo, dependiendo de si el programa de modelado lo permite, también se puede modelar mediante NURBS (Non Uniform Rational B-Spline) [22]. En esta técnica se emplean curvas de Bézier, definidas por puntos de control, como forma básica de modelado, por lo que se obtienen superficies curvadas, al contrario que en el caso anterior, en el que se obtienen formas poligonales. Por lo tanto, se ahorra memoria en comparación a esta, ya que para obtener una superficie curvada lisa tan solo se requieren poco puntos de control. Esta técnica es muy empleada en el diseño de carrocerías de coches y vehículos aeroespaciales. Sin embargo, si la aplicación no soporta este tipo de datos, hay que exportar el modelo a una forma poligonal, perdiendo éste calidad y sirviendo dicha técnica únicamente como ayuda al modelado para crear formas curvas.

Por último, una técnica muy extendida en la animación actual es el modelado por subdivisión. Ésta consiste en crear modelos de alto nivel de detalle a partir de formas simples. De esta forma, el programa de modelado, mediante los algoritmos apropiados, genera geometría adicional a partir de formas simples, por lo que, controlando un pequeño número de polígonos se pueden conseguir geometrías muy detalladas. De la misma forma, una vez se ha creado el modelo con un nivel de subdivisión (y por lo tanto de detalle) apropiado, el programa de modelado lo exporta como una serie de polígonos triangulares para facilitar su manejo en la aplicación.

El número de polígonos que tenga el modelo depende esencialmente del hardware con el que se vaya a manejar sus datos. En el caso de los videojuegos, esto depende de la plataforma o videoconsola donde se haya implementado la aplicación, ya que cada una, dependiendo de su sofisticación técnica, incorpora un chip gráfico más o menos potente capaz de manejar más o menos polígonos. En la Tabla 2.1 se muestra una comparativa del número de polígonos por personaje en función de la plataforma de juego.

PLATAFORMA	POLÍGONOS PINTADOS	Nº POLIS PRINCIPAL	Nº POLIS SECUNDARIOS
Pc/PS3/Xbox360	250 Mp/s 500 M/s	10000 25000	5000 15000
Nintendo Wii	28 Mp/s	5000 15000	2000 7000
Nintendo DS	2048 p/s	200 250	<100

Tabla 2.1: Número de polígonos pintados y por personaje según la plataforma

Adicionalmente, el programa de modelado asigna a cada vértice una serie de propiedades de la superficie, en particular un vector normal a ésta, aproximando la perpendicular a la misma en ese punto. Las normales, como se detallará más adelante, se emplean para calcular información de iluminación: cuanto más paralelo sea el vector de iluminación con la normal, más intenso será el color en ese punto. Aunque, el software de modelado puede importar la información de normales, también pueden ser calculadas por la aplicación en tiempo de ejecución, en el caso por ejemplo de que se modifique la geometría del modelo en esta fase.

2.1.3. Materiales del modelo

Un modelo puede contener tantas mallas como se requiera, pudiendo asignarle a cada una de ellas unas propiedades materiales diferentes. Aquí radica la importancia de las mallas, puesto que a cada una se le puede asignar un material diferente.

A través de la iluminación, se le da una apariencia realista a la escena 3D, dado que es un fenómeno físico muy sensible a la visión humana. Con ésta, se le genera un sombreado sobre los objetos, creando la sensación de volumen. Al igual que en la naturaleza, el color de las fuentes lumínicas y el de los objetos con los que impactan sus rayos determinan el color con el que vemos dicho objeto. Aquí entran en juego los materiales, ya que son los que aportan información acerca de cómo reaccionar ante la iluminación.

Así, los materiales de un modelo son la apariencia que tiene su superficie. Ésta puede ser metalizada, opaca, translúcida, emisiva... cada una de estas apariencias tiene unas determinadas características, moduladas por varios parámetros, que en el caso más simple son: color difuso, color ambiente, color especular y color emisivo. Además a estos parámetros de color hay que sumarle la amplitud del resalte especular y la transparencia. Modificando cada uno de estos parámetros se consiguen apariencias como las mencionadas anteriormente.

Cada uno de los parámetros mencionados corresponde a:

- **color difuso**: es el color reflejado por el material ante iluminación directa (modelo de superficie perfectamente difusora). Es decir que es el color base del objeto, el que define su apariencia básica. Este parámetro es el que predomina y sobre el cual se van a ir modulando los demás parámetros de color para añadir propiedades al material.
- **color ambiente**: es el color reflejado por el material cuando sobre él incide luz ambiente, tan disipada por el entorno que es imposible determinar su dirección. En la mayoría de los casos es muy cercano, sino igual, al color difuso. También se puede entender cómo el color con iluminación muy tenue.
- **color emisivo**: es el color que emite el propio objeto. Cuando un objeto tiene color emisivo, no le afecta la iluminación de la escena, dado que el efecto que se le da al material con este parámetro es que éste genera luz propia.
- **color especular**: es el color de los brillos producidos por el impacto de una luz sobre el objeto. Suele ser blanco o gris, de forma que el color de este brillo sea el propio de la luz incidente. Ajustando el nivel especular, se determina la cantidad de brillos que refleja el objeto y se le da un aspecto más o menos pulido al material.
- **amplitud del resalte especular o *shininess***: este parámetro determina la forma en la que se producen los brillos en el material. De esta forma, se controla la dimensión de los brillos producidos sobre el objeto.
- **transparencia**: es un valor, normalmente comprendido entre 0 y 1, que determina la opacidad del material. Se asignará 1 a objetos totalmente opacos y 0 a objetos completamente transparentes.

2.1.4. Mapeado de texturas

Para terminar de dar detalle al material y dotarlo de una apariencia realista, a éste se le suele aplicar lo que se denomina una textura. Las texturas son imágenes que se "pegan" a la malla o modelo de una determinada forma. Es como si se envolviera el objeto con esta imagen. La forma en la que la textura cubre la malla la determinan las coordenadas de textura.

Las coordenadas de textura son dos valores (u, v) que se asignan a cada vértice de la malla que contiene la determinada textura y que corresponden a un pixel de la imagen, que en este caso se denomina texel. De esta forma, cada vértice se hace corresponder a un punto de la textura y las zonas entre vértices se rellenan con la zona de la imagen entre las coordenadas de textura respectivas. Al proceso de asignación vértice/coordenada de textura se le denomina mapeado de texturas.

Existen dos formas muy comunes de realizar el mapeado. El método más sencillo de cara al diseñador es el de proyección. Consiste en envolver el modelo o una parte de él (subdividiendo el modelo en varios bloques) con un volumen que determina la forma de mapeado. De esta forma, la textura se proyecta, según la superficie del volumen, sobre el objeto. Los volúmenes pueden ser un plano, un cubo, una esfera o un cilindro. Así, un suelo o una pared se mapearía mediante la proyección de un plano, una caja mediante la proyección cúbica, un planeta o una pelota mediante la proyección esférica y una lata de refresco mediante la proyección cilíndrica. Un personaje se podría mapear subdividiéndolo y empleando para cada zona el volumen que más se ajuste a su geometría. Sin embargo, si el volumen de proyección no coincide o difiere mucho de la propia forma del modelo, el mapeado puede resultar muy defectuoso, con repetición de coordenadas de textura sobre una extensa superficie del modelo, situación que ha de evitarse bajo cualquier concepto.

Este proceso puede realizarse en la etapa de diseño, con las herramientas que proporciona el software de modelado, o bien, en ciertos casos puede resultar interesante realizarlo en tiempo de ejecución en el motor de renderizado (del cual se hablará más adelante). En este caso, se calculan las coordenadas de textura para cada vértice sobre la marcha, realizando los cálculos de proyección pertinentes. Esta técnica se emplea para realizar mapeados de entorno, en los cuales se proyecta la imagen de la escena que envuelve el objeto sobre éste y dotarlo del aspecto pulido de los metales, que reflejan los objetos de su entorno. En esta técnica se suelen utilizar o la proyección esférica o la cúbica.

Por otra parte, se puede realizar un mapeado personalizado. Éste se hace en un entorno diferente en el programa de modelado o incluso en un programa independiente. En este entorno se puede modificar la distribución y la forma de la malla con respecto a una textura o mapa de textura, sin por ello modificar la propia geometría del modelo. De esta forma es posible "abrir" (desarrollar) el modelo o malla a lo largo y extenderlo sobre un plano para más tarde hacerlo coincidir con la textura. Como analogía se puede tomar el ejemplo de una lata de refresco, la cual se puede recortar por un lado y desdoblar su superficie para que quede totalmente extendida sobre un plano. También se puede pensar en un cubo que se extiende en su forma de patrón.

Esta forma de mapeado se emplea sobre todo en personajes y modelos más complejos, con diferentes materiales sobre un mismo objeto. Por ejemplo, un personaje hay que texturizarlo de tal forma que las cejas estén bien colocadas en la frente sobre los ojos, que el iris del ojo esté correctamente situado sobre el ojo ocular, o que el dibujo de su

camiseta esté adecuadamente colocado en la parte frontal de la misma.

Un punto importante del proceso de mapeo de texturas es conseguir que la textura quede uniforme sobre el modelo, es decir que los vértices que lo componen se repartan uniformemente sobre la textura. Esto es importante para evitar que en determinadas zonas del modelo la textura quede expandida o estirada o por el contrario pueda quedar contraída, perdiendo resolución. Los efectos mencionados se pueden producir cuando hay una zona donde la densidad de vértices es mayor (esto sucede cuando se quiere dar precisión a la geometría) o menor (en zonas poco detalladas) y no se tiene en cuenta a la hora de crear la textura.

Para evitar estos efectos lo que se suele hacer, mediante las herramientas de mapeado, es o bien, dentro de este entorno, reposicionar a mano los vértices sobre la textura, tarea que puede ser en exceso laboriosa cuando el modelo tiene una cantidad de vértices considerable, o bien estirar o, como se suele denominar, relajar la malla para que los vértices queden uniformemente repartidos sobre la textura.

2.1.5. Texturizado

Una vez generado el mapa de texturas, se obtiene una imagen de plantilla como la mostrada en la Figura 2.1, correspondiente al modelo aún sin texturizar de la Figura 2.2. Como se puede apreciar, se distinguen las mallas del modelo que sirven de referencia para realizar la textura. Además, en este ejemplo la plantilla añade información de volumen a través de la variación de color, que indica la variación de las normales del modelo. Lo más común es llevar la plantilla a un programa de diseño gráfico (por ejemplo Adobe Photoshop) y sobre ésta pintar la textura, teniendo en cuenta la geometría de las mallas y su correspondencia en el modelo.

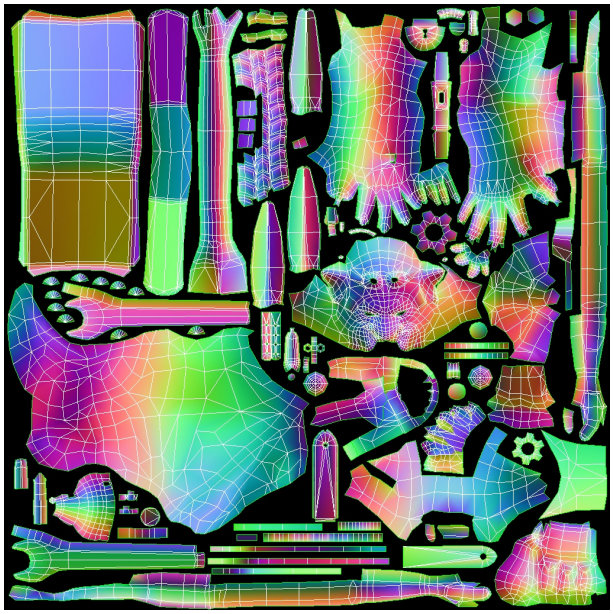

Figura 2.1: Plantilla de textura del modelo.

Este proceso varía en función del grado de detalle que se le quiera imprimir al modelo. En el caso más simple, como es el de los modelos realizados en este proyecto, únicamente se realizará una textura. Ésta será la que dé color al modelo, aportando información al canal

Figura 2.2: Modelo sin materiales ni texturas (modelo cedido por Autodesk).

difuso del material. En el ejemplo anterior, la textura de color (a la que generalmente se le denomina simplemente textura) se puede ver en la Figura 2.3.

Figura 2.3: Textura difusa del modelo.

Sin embargo se le puede añadir complejidad y detalle al modelo aportando información a cada uno de los canales del material (especular, emisivo, brillo...). Siguiendo con el ejemplo anterior, vemos que en primer lugar se la ha añadido una textura o mapa de nivel especular, Figura 2.4. De la imagen en escala de grises se extrae el nivel especular del material en cada punto del modelo correspondiente a la posición en el mapa de texturas,

dada, como ya se ha explicado, por las coordenadas de textura de cada vértice. Negro significa nivel especular nulo y blanco nivel máximo. De esta forma se puede asignar a cada zona del modelo un nivel especular específico. Esto es útil por ejemplo en el caso de un cinturón: la hebilla metálica requiere más nivel especular que un tejido como el cuero, y de esta forma se controla aprovechando un único mapa de texturas.

También se la ha añadido al trol del ejemplo un mapa de color emisivo (Figura 2.5), para controlar qué partes del modelo poseen esta propiedad, así como su color; el mapa del resalte de la amplitud especular de la Figura 2.6, que especifica detalladamente los brillos de cada zona del modelo; un mapa de transparencia (Figura 2.7) que controla la transparencia en ciertas partes del modelo, como las cuerdas, que en la malla son planos gruesos y mediante este mapa se eliminan las zonas sobrantes alrededor de las cuerdas (en este caso blanco es opaco y negro transparente); una máscara que especifica que partes del modelo son piel, para posteriormente aplicarle ciertas características (Figura 2.8).

Figura 2.4: Mapa especular del modelo.

Por último, se ha añadido un mapa de normales (se suele denominar por su nombre inglés normal map o bump map). Este mapa requiere una explicación un poco más detallada. Un mapa de normales suele ser como el que se aprecia en la Figura 2.9, de colores violáceos o azulados. A groso modo, esta imagen lo que contiene es información de normales para cada punto de la superficie del modelo correspondiente a cada píxel de la imagen, en lugar de tener únicamente información de normales para cada vértice. De esta forma se consigue dar a la superficie de la malla una sensación de relieve o rugosidad a través de la iluminación, sin necesidad de añadir polígonos adicionales para conseguir dicho volumen.

En la Figura 2.10 se puede ver un render del resultado final, una vez aplicados materiales, textura y mapas correspondientes a cada canal. Es interesante comparar el anterior render de la malla únicamente y este último para comprobar cómo efectivamente, a través del proceso de texturizado (de forma más o menos compleja) se pueden conseguir resultados muy realistas sin necesidad de modelar una malla de elevada poligonización, ahorrando una cantidad de recursos importante.

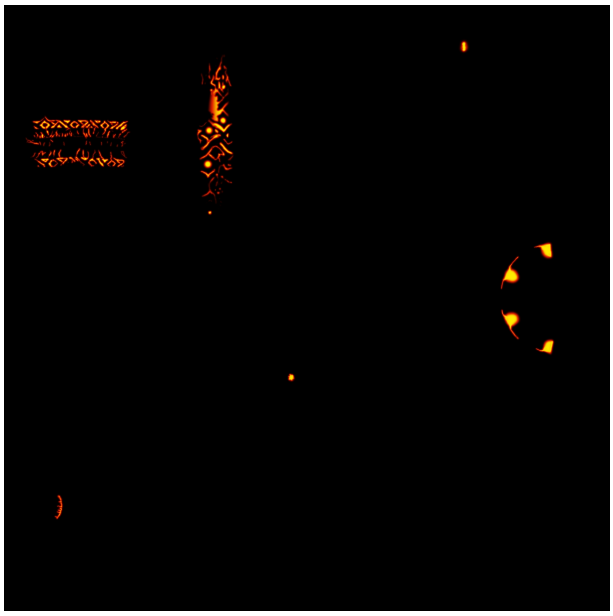

Figura 2.5: Mapa emisivo del modelo.

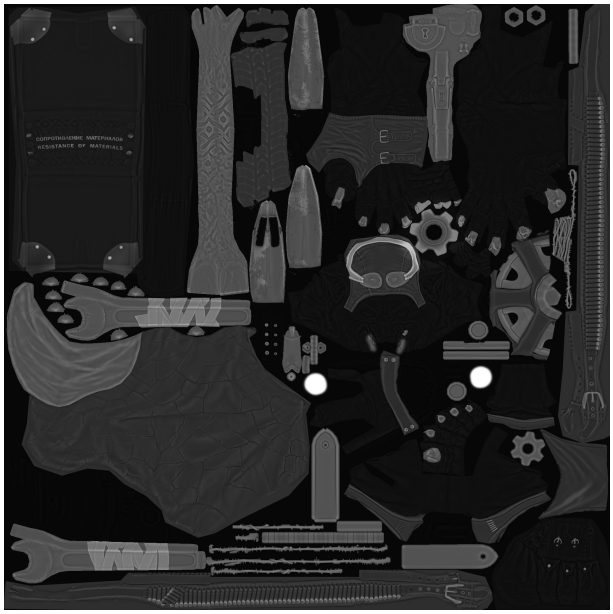

Figura 2.6: Mapa de la amplitud del resalte especular del modelo.

2.2. Animación

La animación es la última etapa de la fase de diseño de los gráficos por ordenador. Es aquí donde se imprime movimiento a los modelos. El proceso es muy similar a la animación 2D tradicional, con la salvedad de que en este caso se trabaja en una escena tridimensional y que basta con mover el objeto o personaje y no hay que redibujarlo en cada frame. Sin embargo sí que existen importantes similitudes y es de hecho recomendable conocer la animación tradicional para realizar animaciones realistas y con "vida", aunque esto es más un tema de diseño que de técnica.

Se puede hablar de dos tipos de animación. La más sencilla, denominada animación de

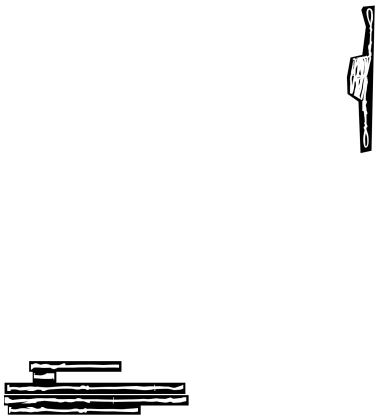

Figura 2.7: Mapa de transparencia del modelo.

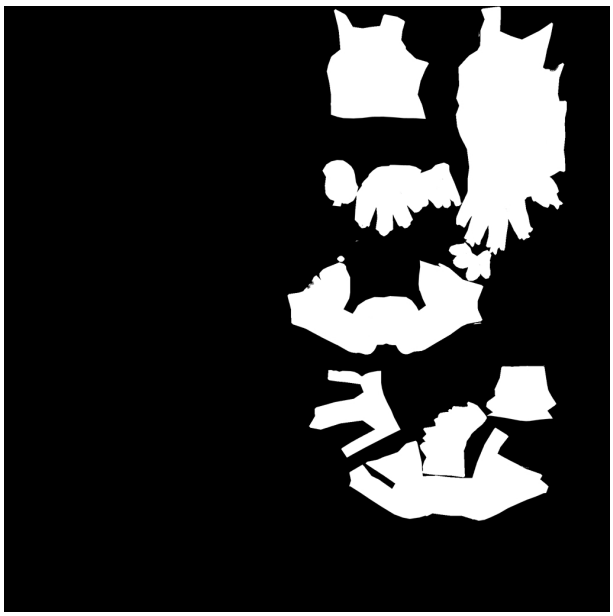

Figura 2.8: Mapa de máscara de piel del modelo.

cuerpos rígidos [1], es la que consiste únicamente en aplicar traslación, rotación e incluso escalado al modelo. De esta manera, la forma original del objeto y la posición relativa de sus vértices queda intacta. Por lo tanto, en la aplicación que maneja la animación tan solo hay que aplicar algoritmos básicos de traslación, rotación y escalado mediante cálculos matriciales. Sin embargo, no se pueden animar objetos orgánicos mediante esta técnica. Si se intentase animar un personaje cuyas articulaciones fuesen bloques independientes (como cilindros) que se mueven independientemente, el resultado sería robótico, puesto que se vería la separación entre cada bloque y no daría la sensación de un ser vivo, con piel.

Con este fin, se emplea la animación por esqueleto [21]. Es un proceso mucho más laborioso

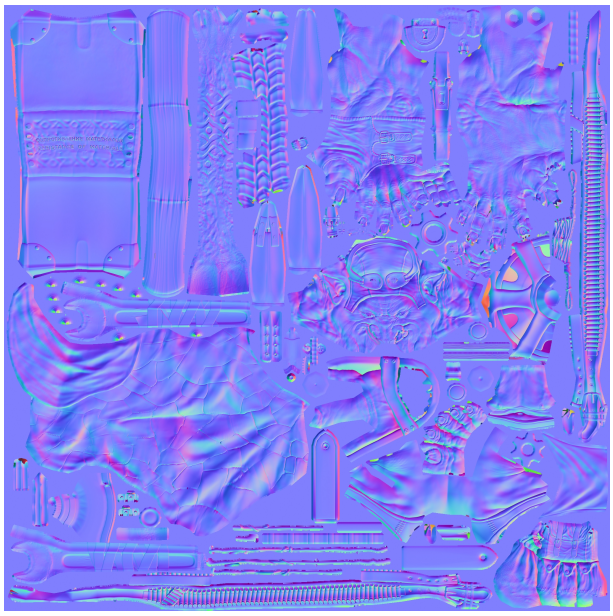

Figura 2.9: Mapa de normales del modelo.

Figura 2.10: Renderizado final del modelo con todos los mapas aplicados.

pero en estas situaciones da resultados más vistosos. La idea es asociar al modelo un esqueleto compuesto por varios bloques o huesos, unidos por articulaciones, con una jerarquía entre cada una de ellos. De esta forma, el torso controla al antebrazo, que a su vez controla al brazo, que controla la mano que a su vez controla los dedos, y así sucesivamente con cada extremidad, del centro hacia fuera. Por supuesto, al esqueleto se le puede dar el detalle que se precise (un dedo puede ser un único hueso o puede estar compuesto por tres para poder abrir y cerrar la mano de forma realista). Con esta jerarquización se consigue que al mover una articulación, los huesos hijos asociados

acompañen el movimiento (de la misma forma que si se levanta el brazo a partir del hombro todo el brazo se eleva, y no sólo el antebrazo).

Una vez construido el esqueleto y realizada la jerarquización, se limitan las rotaciones de las articulaciones. Es necesario para impedir que por ejemplo el codo se doble hacia atrás (parecería una rotura) o que la cabeza atraviese el torso. Así se puede manipular el esqueleto con la seguridad de que nunca se producirán estas aberraciones anatómicas.

Tras configurar el esqueleto se le asocia la malla del modelo, como si ésta fuese la piel flexible del esqueleto, que es lo que realmente se anima (de ahí que en inglés se denomine a esta técnica de animación *skinned animation* o *skeletal animation*). Así, a cada hueso se le asignan unos vértices, que se moverán con el movimiento de éste. No obstante, un vértice puede ser asociado a más de un hueso, de tal forma que su movimiento lo controlen varios agentes (esta es la verdadera trascendencia de ésta técnica, dado que en realidad, la animación de cuerpos rígidos también se puede conseguir mediante esqueleto). La cantidad de movimiento que un hueso ejerce sobre un vértice lo determina un coeficiente llamado peso. Dada la naturaleza de este coeficiente, la suma de los pesos de un vértice es igual a 1. La asignación de los pesos se realiza de tal forma que cuanto más próximo esté un vértice de una articulación, más se equilibran los pesos entre el hueso principal sobre el que reposa el vértice y el contiguo. De esta forma, al rotar una articulación, los vértices más próximos a esta no acompañan tanto el movimiento del hueso que lo realiza y se aproxima más a un movimiento orgánico o muscular, dotando de flexibilidad a la malla. Con el modelo perfectamente "riggeado" (a este proceso en inglés se le llama *rigging*) ya se puede empezar a animar el modelo. En la Figura 2.11 puede el resultado final de un modelo riggeado.

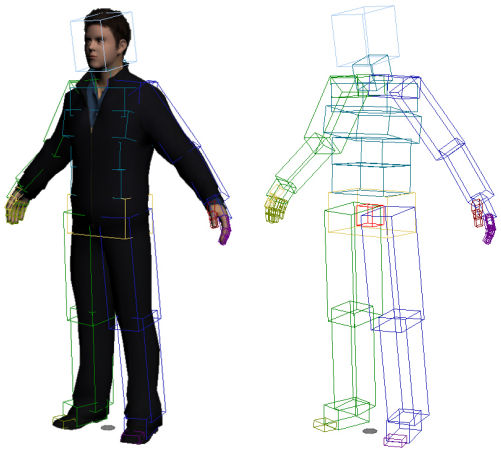

Figura 2.11: Modelo con esqueleto asociado (izquierda) y esqueleto solo (derecha).

Tanto si se emplea una técnica u otra, el proceso de animación es el mismo. Consiste básicamente en definir unas claves de animación en las cuales se le asigna al modelo una transformación o una pose, como se haría en la animación tradicional con las imágenes en cada frame. Sin embargo, a diferencia de ésta, no es necesario definir una clave por frame, sino que se definen claves separadas en el tiempo, y el propio software se encarga de interpolar el movimiento entre claves. Dependiendo del programa, la interpolación puede ser predefinida (lineal) o bien configurable a través de la definición de curvas de velocidad para cada canal de transformación (posición en (x, y, z), rotación en (x, y, z), escalado en (x, y, z)). De esta forma se puede controlar la caída de una pelota

simplemente definiendo el principio y el choque contra el suelo y mediante una curva exponencial simular la aceleración de la pelota.

2.3. Renderizado

El renderizado (del inglés render) es el proceso bajo el cual, a partir de un modelo o entorno tridimensional, se genera una imagen bidimensional. Esta imagen se mostrará posteriormente en algún tipo de dispositivo de visualización, véase un monitor de ordenador, una televisión, o en el caso de la Nintendo DS sus pantallas LCD.

El renderizado es un proceso clave dentro de los gráficos por ordenador, pues es la capa que se encarga de transportar el modelo 3D previamente generado a una imagen 2D visible, con mayor o menor fidelidad y detalle. Por ello, aunque se disponga de unos modelos muy bien construidos y detallados, si el motor de renderizado no es bueno, el resultado final no será satisfactorio (y a la inversa ocurre lo mismo).

Además, en esta fase es en la que se genera una escena más o menos realista alrededor de los objetos poligonales. Es decir, que simula las condiciones físicas tales como la iluminación o las leyes ópticas de visualización, que dependiendo del grado de complejidad de los algoritmos bajo los cuales se calculen estas propiedades, la escena tridimensional visualizada será más o menos acorde al mundo real, será más o menos fotorrealista.

La importancia del tiempo de renderizado varía según la aplicación. En el caso de las películas de animación, el renderizado puede ser considerablemente más lento debido a que se generan una cantidad fija de imágenes (que depende de la duración del film), para posteriormente ser reproducidas a una tasa de cuadros por segundo fija de 24 imágenes por segundo, simulando movimiento con la sucesión de fotogramas. Sin embargo, la velocidad a la que se generan dichas imágenes no es tan importante (siempre dentro de los límites), dado que solo se calculan una vez y luego se almacenan. Por ello, el renderizador puede manejar más cantidad de polígonos, texturas de mayor calidad o realizar cálculos de iluminación más complejos, por poner algunos ejemplos. Esto quiere decir que una imagen puede tardar varios segundos en ser calculada. Tanto es así, que no es raro que el renderizado de toda la batería de imágenes de la animación completa tarde días o incluso semanas en completarse. Por esta razón, tradicionalmente, la vanguardia de gráficos por ordenador siempre se ha movido en este campo, dado el margen de tiempos con el que se trabaja.

Sin embargo, en aplicaciones en tiempo real como los videojuegos, simuladores o ciertas aplicaciones médicas, es imprescindible tener bien controlado el tiempo de renderizado [1]. En este caso hay que procesar las imágenes en tiempo de ejecución y manejar la interacción con el usuario, además de realizar otros cálculos aparte como la actualización de entidades, la inteligencia artificial, el sonido, las físicas, etc. Un concepto a tener muy en cuenta es que, a diferencia de las animaciones pre-renderizadas, en este caso no solo basta con mostrar las imágenes y generar un efecto de movimiento constante, sino que interviene el concepto de interacción. A partir de unos 6 fotogramas por segundo (fps) la sensación de interacción empieza a aparecer. A 15 fps se puede empezar a hablar de tiempo real. En este tipo de aplicaciones un valor mínimo se establece en 30 fps (dependiendo del campo de aplicación este mínimo puede variar), aunque lo ideal es llegar a los 60 fps para alcanzar una fluidez óptima. Según ciertos estudios, a partir de unos 72 fps las diferencias

de la tasa de imágenes son prácticamente indetectables.

Esto limita drásticamente la cantidad de cálculos que se pueden realizar por ciclo de render (por imagen), ya que hay que realizar un mínimo de ciclos por segundo para mantener la sensación de movimiento y de interactividad. De hecho éste es uno de los trabajos fundamentales de las tarjetas gráficas (llamadas GPUs, *Graphics Processing Units*): realizar los cálculos específicos de gráficos mediante procesos y algoritmos propios y así quitarle trabajo a la CPU del ordenador y que ésta se pueda encargar de realizar otros procesos.

Por ello, las capacidades del renderizador dependen en gran medida del procesador de gráficos sobre el que trabaje. Cuanto más rápido sea y más cálculos pueda realizar, más rápido se generarán las imágenes correspondientes a las variaciones de la escena 3D. Sin embargo también influyen la memoria que tenga para almacenar información visual, si incorpora varios procesadores con los que realizar diferentes cálculos simultáneos o simplemente la arquitectura del hardware, de forma que sea más o menos eficiente. Estas capacidades, lógicamente, van aumentando y mejorando a lo largo de los años, por lo que no es comparable lo que se puede hacer con el chip gráfico de la Nintendo DS, por poner un ejemplo, a lo que se consigue con las tarjetas gráficas actuales.

Excusa decir que el hecho de tener que calcular una cantidad elevada de imágenes por segundo provoca que la calidad y el detalle gráfico en estas aplicaciones nunca llegue al de las animaciones pre-renderizadas, ya que con la misma tecnología del momento es inviable.

En este documento nos centraremos principalmente en la arquitectura de los renderizadores en tiempo real, que son los que nos ocupan.

Así, se comentará de forma general el proceso de renderización y el llamado *pipeline* de render. La Figura 2.12 muestra un diagrama de dicho proceso. Este sistema ha variado poco desde los comienzos con los primeros motores de render, si bien se le va añadiendo mejoras, con técnicas y algoritmos cada vez más complejos, acordes a las capacidades tecnológicas y computacionales del momento. Evidentemente, con cada avance de la tecnología informática se pueden procesar más polígonos de forma más rápida y realizar cálculos de iluminación más complejos, además de emplear técnicas más avanzadas de procesado de imagen, cómo veremos más adelante.

No se ahondará sin embargo demasiado en detalle ya que no es este el tema de estudio, sino un emplazamiento en el estado del arte para, una vez metidos en materia en lo referente a la Nintendo DS, el lector sea capaz de situarse y tenga unos conocimientos básicos, además de entender las limitaciones de dicha consola.

2.3.1. Etapa de aplicación

La etapa de aplicación es completamente implementada por el usuario. Esta fase, dependiendo de los algoritmos empleados puede, ralentizar todo el proceso subsiguiente, de modo que es imprescindible que esta etapa sea eficiente si se quiere que sacar provecho del 100

La aplicación 3D, en este caso el motor de render programado por el usuario, lanza ciertos comandos relativos a la geometría a la API gráfica (en desarrollo para un ordenador hablaríamos de OpenGL [19] o Direct3D [9], en el caso de la Nintendo DS se ha utilizado

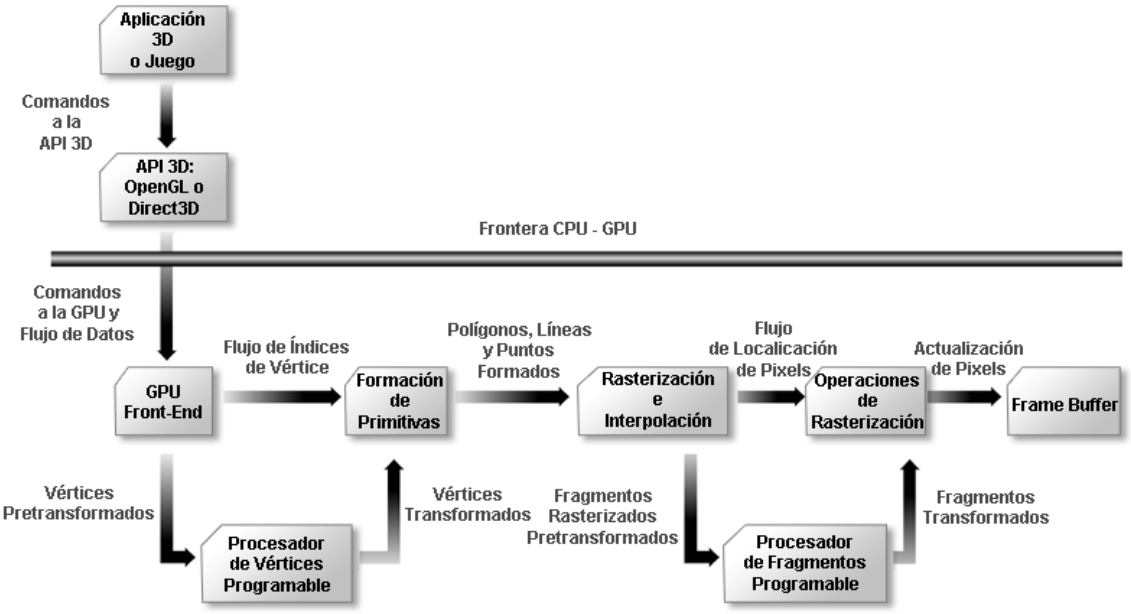

Figura 2.12: Cadena de procesamiento del renderizado de gráficos en tiempo real.

como se mencionará más adelante Libnds de DevkitPro [8]). Esta capa es la que se encarga de comunicarse con el hardware, la tarjeta gráfica, traduciendo dichos comandos en llamadas a los registros necesarios para ejecutar cada acción.

De esta forma, al final de la etapa de aplicación la geometría que debe ser renderizada pasa a la fase de geometría. Ésta corresponde a las llamadas primitivas de render, es decir, puntos, líneas, triángulos y en algunos sistemas cuadriláteros, que en según qué casos terminarán pintadas en pantalla. Este proceso es el más importante de la etapa de aplicación.

Sin embargo, también se realizan otras tareas como la detección de colisiones y las físicas (que repercute en la transformación de la geometría), la animación por transformación, la animación de texturas o la lectura y procesamiento de dispositivos externos como ratón y teclado o controlador de juegos y su influencia en la aplicación (como el movimiento de un personaje).

En la Figura 2.13 se muestra un esquema típico del bucle de un ciclo de juego que ilustra el proceso de una aplicación de render en tiempo real, más concretamente de un videojuego (aunque es aplicable a otras ramas).

2.3.2. Etapa de geometría

La etapa de geometría es la que se encarga de realizar los cálculos de vértices y de polígonos. Se divide en varios sub-procesos que pueden ser completa o parcialmente programables por el usuario o pueden no serlo en absoluto. Éstos son: transformación de modelo y de vista, operaciones sobre vértices (vertex shading), proyección, clipping y mapeo de pantalla. En la Figura 2.14 se muestra un esquema de dicho proceso.

Los datos de geometría básicos son los vértices. Como ya se comentaba anteriormente éstos van asociados a cierta información: coordenadas (x, y, z) que determinan su posición

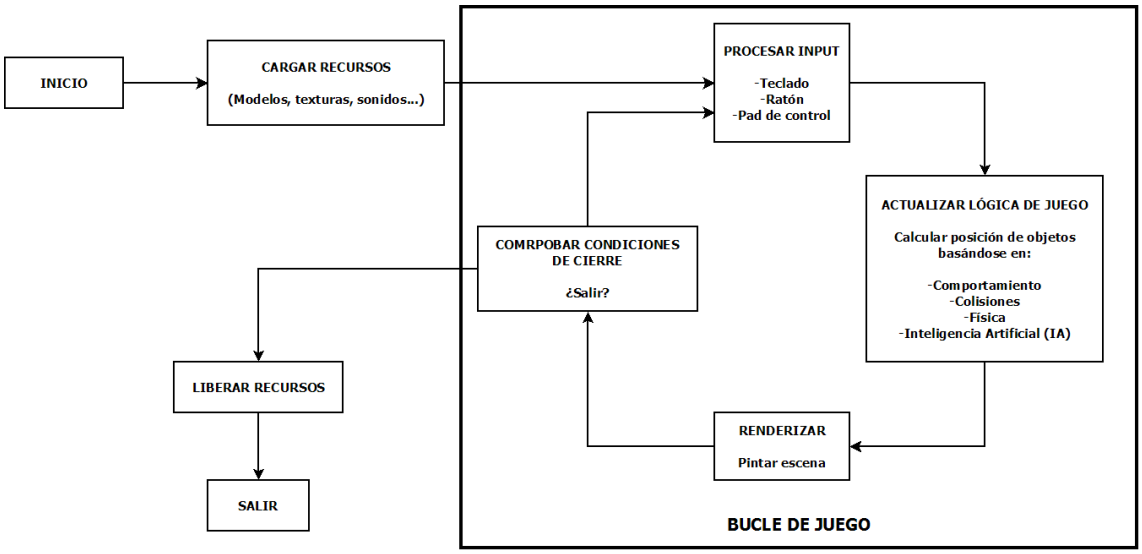

Figura 2.13: Esquema general típico de un ciclo de juego.

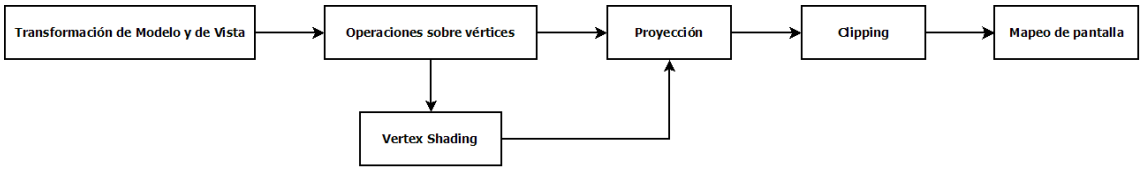

Figura 2.14: Esquema de la etapa de geometría del proceso de render, dividida en varios sub-procesos.

en el espacio y sus coordenadas de textura (u, v), así como una normal asociada.

En el primer sub-proceso de la etapa de geometría cada vértice de un objeto (y por lo tanto el objeto en sí) tienen una posición en el **espacio de mundo**. Esta posición se obtiene aplicando a las coordenadas del objeto la transformación de modelo. La transformación de modelo va asociada a cada uno de los modelos en escena y un modelo puede tener asociadas varias transformaciones de modelo con el fin de crear instancias independientes de dicho objeto. De esta forma, cada objeto tiene una posición en el espacio único de mundo.

Los objetos que se pintarán en pantalla son aquellos a los que la cámara virtual enfoca. La cámara tiene su propia posición y rotación en el espacio de mundo, así como otras propiedades que repercuten en el volumen que determina qué objetos son representados, llamado *frustrum*. Pues bien, la **transformación de vista** posiciona la cámara en el origen de coordenadas y enfocando en el eje z negativo (suponiendo que éste es el correspondiente a la profundidad) y reposiciona el resto de objetos con respecto a ésta. Este proceso se realiza con el fin de facilitar las siguientes operaciones de proyección y *clipping*.

Como ya se ha comentado en apartados anteriores, para conseguir un aspecto realista cada objeto debe de tener unas propiedades de material que determinan cómo le afecta la iluminación de la escena. El proceso de realizar dichos cálculos de iluminación se denomina *shading*. Aunque los cálculos sobre vértices no siempre han sido programables (en la Nintendo DS tan solo son configurables, no admite *shaders*), éste ha sido uno de los grandes avances en gráficos por ordenador en tiempo real. Esto permite inyectar programas independientes llamados *shaders* (en este caso *vertex shaders*) que ofrecen

control total sobre cómo han de tratarse los materiales de un modelo en una escena iluminada. En esta fase intervienen las normales asociadas a cada vértice, indispensables en los cálculos de iluminación.

El siguiente sub-proceso de la etapa de geometría es la proyección. Aquí es donde se determina y se aplica la perspectiva que se le quiere dar a la escena. La proyección transforma el volumen de vista en el cubo unitario, cuyos extremos son (-1, -1, -1) y (1, 1, 1), que se proyectará sobre la imagen de pantalla. Esta transformación se realiza mediante la matriz de proyección. Existen dos formas principales de proyección: la ortográfica y la de perspectiva.

En la proyección ortográfica el volumen de vista es rectangular y la imagen proyectada de un objeto no varía en función de la distancia a la cámara, ya que las líneas de proyección son paralelas. Esto se traducen en que los objetos en la escena mantienen sus dimensiones independientemente de su profundidad o distancia a la cámara, además de que las líneas paralelas quedan en todo momento paralelas. Esta proyección, aunque no muy realista, se emplea en cierta clase de juegos donde el jugador tiene visión y control global de la acción. Es una especie de visión omnisciente o divina.

Por el contrario, en la proyección de perspectiva el volumen de vista (*frustrum*) es una pirámide truncada con base rectangular. Esto produce que cuanto más alejado esté un objeto del punto de visión, más pequeña será su proyección sobre el plano de imagen. En este modo, las líneas paralelas convergen en un punto.

Una vez se ha determinado el *frustrum* y se ha transformado al cubo unitario, el sistema de render determina qué objetos se visualizan y cuales se pueden descartar en el pintado. Esto se realiza determinando qué objetos están dentro del *frustrum*, cuáles lo están parcialmente y cuáles no lo están en absoluto. Si un objeto está íntegramente dentro del *frustrum* éste se manda a la siguiente fase tal cual. En el caso de que el objeto no esté contenido dentro del volumen de vista se desecha para ahorrar tiempo de procesado en los pasos posteriores. Finalmente, si el objeto está parcialmente contenido dentro del *frustrum* se modifica su geometría para que ésta esté completamente dentro del *frustrum*. Para ello, en el punto donde el plano del frustrum corta el objeto, se crean los vértices necesarios que cierran la geometría del objeto y se desechan los antiguos vértices.

El último proceso de la etapa de geometría es el mapeo de pantalla. En esta fase, las coordenadas x e y de cada primitiva dentro del cubo unitario se transforman a coordenadas de pantalla mediante un escalado dependiente de las dimensiones de la ventana de la aplicación. Las coordenadas de pantalla junto a las coordenadas z (que no se transforman) forman las coordenadas de ventana que pasarán a la etapa de rasterizado para convertir la escena 3D en una imagen bidimensional.

2.3.3. Etapa de rasterizado

El rasterizado es el proceso a través del cual se digitalizan las primitivas de geometría de la etapa anterior, generando los puntos de imagen correspondientes en la pantalla de visualización. A esta etapa llegan los vértices transformados y proyectados junto con la información de sombreado (generada en el shader). Así pues, en esta fase el objetivo es calcular el color de cada pixel, teniendo en cuenta esta información de geométrica y de color.

Tal y como en la etapa de geometría, este proceso se divide en sub-procesos, los cuales algunos son programables, otros configurables y otros se ejecutan de forma cerrada. Estos procesos son: configuración de triángulos, recorrido de triángulos, pixel shading y mezcla (blending). En la Figura 2.15 se muestra un esquema de dicho proceso.

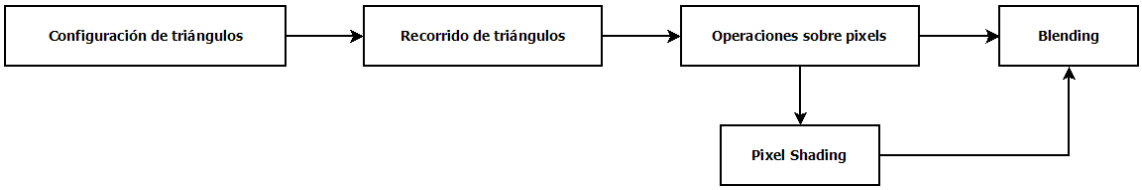

Figura 2.15: Esquema de la etapa de rasterizado del proceso de render, dividida en varios sub-procesos.

En la fase de configuración de triángulos, se prepara la información de los triángulos, tanto para ser pasada al siguiente proceso, como para manejar los posibles diversos datos de sombreado generados en la etapa de geometría. En este punto se conectan los vértices de las distintas primitivas. La conexión se realiza según la topología que se haya especificado: triángulo como lista (se especifican 3 puntos y se conectan uno tras otro y se pasa a otra entidad independiente), triángulo en banda (se especifican un número N de puntos y se forman triángulos aprovechando los dos puntos anteriormente dados) o triángulo en abanico (se especifican N puntos y se forman triángulos tomando como punto de inicio el primer punto especificado).

Tras este proceso se realiza lo que en ingles se denomina *back face culling* o desecho de las caras traseras en castellano (todavía en la etapa de configuración de triángulos). Esta técnica se emplea para deshacerse de los polígonos que estén de espaldas a la cámara porque estén tapados por polígonos de cara a la cámara y así agilizar la rasterización. Para determinar qué caras están de espaldas a la cámara se tiene en cuenta el orden de pintado de los vértices que determina el vector normal: si se pintan en orden horario, la normal calculada apunta hacia la cámara y por lo tanto el polígono está de frente a ésta; si por el contrario el sentido es anti-horario, la normal apuntará en sentido opuesto a la cámara y el polígono está de espaldas a ésta y por lo tanto, si el *back face culling* está activado, éste se desechará. En ciertas ocasiones, por ejemplo cuando se desea pintar un objeto no sólido, es necesario deshabilitar el *back face culling*, ya que de lo contrario no se renderizaría correctamente. Los triángulos no desechados se mandan al proceso de recorrido de triángulos.

En el siguiente sub-proceso se hace un barrido de los triángulos y se comprueba qué píxels se encuentran cubiertos por un triángulo, generando lo que se denominan fragmentos. Se crean de esta forma fragmentos para cada píxel que cubre el triángulo. Los fragmentos entre vértices se generan interpolando la información de los vértices del triángulo (ya sea su posición, su color, sus coordenadas de textura...). De esta forma, cada fragmento contiene como mínimo estos datos:

- Posición del píxel (x, y). Rango: ([0, ancho de pantalla], [0, alto de pantalla]).
- Profundidad (z)
- Datos de vértice adicionales (color, coordenadas de textura, vector normal...).

Una vez creados los fragmentos correspondientes a cada triángulo estos se envían al *pixel shader*, dónde se realizan operaciones de píxel. Como ya se ha comentado, los *shaders* son

programas independientes completamente programados por el usuario, lo que confiere una versatilidad y un control totales. Este proceso tiene como objetivo calcular el color final de cada fragmento. Las operaciones más comunes en los *pixel shaders* son las de texturizado y los cálculos de iluminación por píxel. Es en este punto donde se aplican las propiedades extraídas de los mapas adicionales (especular, de brillo, emisivo, de normales...), cuya generación y funcionalidad se comentaron en el apartado de texturizado, además de la textura básica.

El texturizado consiste en una serie de algoritmos que extraen las coordenadas de textura para cada fragmento, teniendo en cuenta las de los vértices que componen los triángulos, y aplican el color de dicho texel al fragmento considerado. Esto se realiza para cada mapa de texturas, comentados en el apartado de modelado.

Hay ocasiones en las que las coordenadas de textura obtenidas se salen del rango [0, 1]. En estos casos se aplican diferentes técnicas para texturizar por completo en objeto:

- **Repetición**: la textura se repite a lo largo del objeto. Es útil por ejemplo si se quiere texturizar un muro de ladrillos: de esta forma no es necesario pintar un número excesivamente elevado de ladrillos, basta con pintar una textura de unos pocos y repetirla sobre todo el objeto.
- **Espejo**: es parecido a la repetición, con la diferencia de que la textura, una vez se repite, se voltea para formar un reflejo con la anterior.
- **Fijar**: valores fuera del rango [0, 1) se fijan a dicho rango. Esto se traduce en la repetición de los bordes de la textura.
- **Borde**: valores fuera del rango [0, 1) se renderizan con un color de borde definido.

El último proceso, tanto de la etapa de rasterizado como del pipeline es la de mezcla o *blending*. Su objetivo es combinar el color presente en el *color-buffer* (array de colores con las componentes r, g, b de cada color), con el color calculado en el *pixel shader*.

Además, en este proceso se resuelve la visibilidad de los fragmentos de la escena. La profundidad de cada fragmento renderizado se compara con la del píxel almacenado en el Z-buffer o *depth buffer*. Si la profundidad de éste es menor que la del pixel almacenado, se pinta este fragmento en el pixel y se reemplaza en el Z-buffer, de lo contrario el fragmento no se pinta. En el caso de pintar superficies translúcidas es necesario pintar los fragmentos opacos primero y de atrás a adelante, si no éstas no se renderizan correctamente.

Una comprobación adicional se hace con el *alpha test*: se comprueba el alfa de un fragmento dado. Si éste no pasa el test especificado (alfa mayor, alfa igual...) el fragmento no se pinta y no se tiene en cuenta en el resto del proceso. Se realiza el *alpha test* para asegurarse de que un fragmento totalmente transparente no afecte al Z-buffer.

Además, las tarjetas gráficas modernas disponen del llamado *stencil buffer*. Éste sirve para realizar efectos especiales. Se almacenan primitivas en este buffer con el fin de controlar el *color buffer*. De esta manera, por ejemplo, si se pinta un círculo "relleno" en el *stencil buffer*, con un algoritmo se podría controlar que solo se pinten las primitivas en el *color buffer* allá dónde el círculo esté presente.

La tarea final de esta fase es hacer el *blending* o la mezcla de los colores en cada píxel, como por ejemplo el de un objeto translúcido mezclado con el color de un objeto opaco que esté detrás.

Una vez hecho este proceso la información de color de cada píxel se envía al *frame buffer*

que almacena la imagen en cada momento para ser enviada al dispositivo de visualización.

2.4. Comparativa con la Nintendo DS y conclusiones

El desarrollo de gráficos por ordenador es un proceso muy complejo en el que intervienen disciplinas tan dispares como el diseño y la programación. Además, a medida que los equipos de desarrollo crecen, estos campos se especializan cada vez más, independizándose el modelador del texturizador y del animador (incluso dentro de éstos hay modeladores de personajes y modeladores de entorno, animadores de juego y de escenas de vídeo en el caso de los videojuegos...), y el programador del motor de juego del de herramientas y el de lógica, debido a esta complejidad creciente.

Sin embargo, esto es diferente cuando se trabaja en una plataforma cerrada, en este caso la videoconsola Nintendo DS, que además tiene más de 8 años. Tecnológicamente hablando este lapso de tiempo es gigantesco y más si tenemos en cuenta que la máquina es portátil, por lo que ya en su época no reflejaba los mayores avances en este ámbito.

Todo el proceso general explicado a lo largo del capítulo es aplicable al desarrollo en Nintendo DS. Sin embargo se pueden establecer unas diferencias y unos límites claros con respecto al desarrollo en las máquinas punteras actuales. Aunque se enumerarán ciertas diferencias y limitaciones de la consola, no se darán datos concretos ya que las especificaciones de ésta se detallarán más adelante.

La primera limitación destacable de la Nintendo DS frente a máquinas más potentes es que dispone de una memoria muy reducida, tanto la principal como la de video. El hecho de que la memoria principal sea escasa (del orden de los megabytes, frente a los gigabytes que poseen actualmente las tarjetas gráficas) se traduce en que se pueden manejar datos más pequeños, y menor cantidad de ellos. Por ejemplo, se pueden manejar menos modelos por escena, y hay que controlar el tamaño de las estructuras que los contienen, ya que una mala gestión en este aspecto resultaría en escenas desiertas, poco detalladas y sin vida (no es lo mismo una habitación con muebles y objetos cotidianos que una sala con cuatro paredes y una sola mesa en el centro).

Con la memoria de video (VRAM) ocurre lo mismo: es muy limitada (del orden de los kilobytes) y provoca que no se puedan cargar en ella más que unas pocas texturas, de tamaño reducido (perdiendo detalle) si se quiere manejar más cantidad de ellas. Además, el tamaño máximo de las texturas que la Nintendo DS puede manejar es también reducido (alrededor de 4 veces menor que en sistemas actuales, en los cuales se manejan texturas de alta definición). Estos aspectos reducen el margen de maniobra tanto para el diseñador, que tiene que crear modelos vistosos de baja poligonización y texturas de pequeñas dimensiones, como para el programador, que tiene que almacenar los datos reduciendo al máximo la memoria requerida.

Otro aspecto técnico limitado es la velocidad del procesador. Obviamente, con el paso de los años la tecnología en este campo se queda en seguida obsoleta y en este caso también existen diferencias muy grandes (la velocidad del ARM9 de la Nintendo DS es del orden de los megahercios, mientras que las tarjetas gráficas actuales trabajan con órdenes de magnitud del gigahercio).

Otro aspecto que diferencia al chip gráfico de la Nintendo DS es su limitación del número máximo de vértices/triángulos por pantalla. Esto no se produce en sistemas actuales,

donde este aspecto no viene limitado explícitamente, sino que depende de la velocidad media de refresco que se requiera (a mayor velocidad, menor manejo de datos y por lo tanto menor carga de polígonos), de la eficiencia del motor gráfico, etc. Esto se debe quizás a que la Nintendo DS trabaja siempre a 60 fps, independientemente de los cálculos que se realicen por ciclo, y por lo tanto se tiene que asegurar de que nunca se va a intentar pintar más de lo que puede procesar.

Por otra parte, también existen diferencias en cuanto al modo de trabajar el aspecto visual que se reflejan en la reducción de la calidad gráfica. Por un lado, la Nintendo DS no permite el filtrado de texturas. Esto provoca que las texturas en ciertas ocasiones se vean en exceso pixeladas, o bien, que aparezcan artefactos indeseados en texturas con altas frecuencias, es decir, variaciones muy rápidas del nivel de brillo, que generan patrones de Moiré de baja frecuencia muy visibles y molestos [23]. Por otro lado, la consola no soporta los shaders (tan solo las tarjetas relativamente modernas trabajan con ellos). Esto implica que no se pueden realizar ciertos efectos visuales ni se pueden emplear los mapas de textura adicionales, reduciendo la calidad y el detalle visuales. Sin embargo, como se detallará más adelante, se pueden simular ciertos efectos mediante otras técnicas como la utilización sucesiva de varias texturas, o los colores por vértice en conjunción con el texturizado.

La Nintendo DS es por lo tanto un sistema, lógicamente, limitado y cuyo desarrollo difiere en varios puntos concretos con el de los chips actuales. Sin embargo, el proceso general es muy similar y en él puede intuirse la forma de trabajar en las primeras arquitecturas gráficas. De esta forma, para sacar partido al motor 3D de la Nintendo DS es necesario tener en cuenta estas limitaciones e implementar en consecuencia cada unos de los recursos necesarios para hacer que el juego resulte lo más vistoso posible. Éste será por lo tanto el objetivo de este proyecto: estudiar las características del motor 3D de la consola y experimentar hasta dónde es capaz de llegar en términos gráficos.

Capítulo 3

Nintendo DS

La Nintendo DS es una videoconsola portátil desarrollada por la compañía japonesa Nintendo. Salió al mercado en el año 2004 (wiki, nintendo). La particularidad de la consola radica en la inclusión de dos pantallas LCD, siendo una de ellas táctil. El uso de una pantalla táctil, que a día de hoy parece una obviedad, dada la proliferación de smartphones y tablets, en 2004 era una auténtica novedad, siendo de los primeros dispositivos en incorporar esta tecnología.

Gracias a esta incorporación, Nintendo dio un giro a la industria del videojuego. Por una parte, gracias a la novedad de las dos pantallas y la pantalla táctil, los desarrolladores se centraron en hacer videojuegos con controles y mecánicas más accesibles que aprovecharan estas características, pudiendo así desviar la atención de los gráficos. De esta manera, se optó por reducir la potencia gráfica de la consola en pos del abaratamiento de ésta. Por otro lado, el hecho de desarrollar juegos que se alejaban del estándar "tradicional" y que aportaban por el contrario más retos mentales (Brain Training o Profesor Layton), entretenimiento a las niñas (saga Imagina ser. . .) o incluso aprendizaje (PaintAcademy) produjo un acercamiento masivo de los videojuegos a mercados hasta entonces inaccesibles, a otros usuarios potenciales que no eran jugadores habituales.

Debido a la accesibilidad de la consola y al desarrollo de un abanico de juegos para toda la familia, la Nintendo DS se convirtió en un rotundo éxito de ventas, siendo a día de hoy la videoconsola más vendida de la historia, por encima de potentes máquinas como Xbox 360, de Microsoft, o Playstation 3 y Playstation 2, de Sony (ésta última siendo la segunda más vendida).

Además del éxito de ventas, la Nintendo DS se ha convertido en un éxito entre los desarrolladores aficionados gracias a la facilidad de desarrollar y ejecutar homebrew. Como se detallará en capítulos posteriores, la creación de herramientas externas al SDK de Nintendo por parte de aficionados y la posibilidad de usar flashcards la han convertido en una plataforma idónea para la creación de juegos "caseros". De hecho, esta es una de las motivaciones para la realización de este proyecto, que de otra forma no podría haberse llevado a cabo.

3.1. Especificaciones técnicas del hardware

Las especificaciones técnicas de la Nintendo DS han sido extraídas directamente de gbatek [10]. En algunos casos se especifica el valor de cierta característica en otras consolas portátiles tales como GameBoyAdvance (GBA), la precursora de la Nintendo DS, y PlayStation Portable (PSP) de Sony, competidora directa, para tener una referencia.

- Procesadores
  - ARM946E-S 32 bit RISC CPU, 66 MHz (PSP: CPU MIPS R4000 de 32 bit a 333 MHz, con un sub-procesador de gráficos de 166 MHz, 32bit)
  - ARM7TDMI 32 bit RISC CPU, 33 MHz (GBA: ARM7 32 bit RISC CPU, 16,78 MHz)
- Memoria interna
  - 4096 kB RAM Principal (GBA: 288 kB; PSP: 32 – 64 MB dependiendo del modelo)
  - 96 kB WRAM
  - 60 kB TCM/Cache
  - 656 kB VRAM (asignada a BG/OBJ/2D/3D/Paletas/Texturas/WRAM) (GBA: 96 kB; PSP: 2 MB)
  - 4 kB OAM/PAL (2K OBJ Attribute Memory, 2K RAM de paletas estándar)
  - 248 kB Memoria interna 3D (104K RAM de polígonos, 144K RAM de vértices)
  - 8 kB RAM de Wifi
  - 256 kB Firmware FLASH
  - 36 kB ROM BIOS (4K NDS9, 16K NDS7, 16K GBA)
- Video
  - 2pantallas LCD (cada una de 256x192 píxels, 3 pulgadas, 18 bits de profundidad de color y retro-iluminadas)
  - 2 motores de video 2D
  - 1motor de video 3D (puede ser asignado tanto a la pantalla superior como a la inferior)
  - 1 capturador de video (para efectos o para superponer 3D sobre el segundo motor 2D)
- Audio
  - 16 canales de audio
  - 2 unidades de captura de audio (para efectos de eco, etc)
  - Salida: 2 altavoces estéreo integrados y entrada de auriculares
  - Entrada: 1 micrófono integrado y entrada de micrófono
- Controles
  - 4 teclas de dirección, 8 botones (A, B, X, Y, L, R, SELECT y START)
  - Pantalla táctil (en pantalla inferior)

- Puertos de comunicación
  - Wifi IEEE802.11b
- Memoria externa
  - Ranura NDS (para juegos de NDS)
  - Ranura GBA (para expansiones de NDS, o para juegos de GBA)
- Cartuchos de fábrica
  - ROM: 16MB, 32MB, 64MB, 128MB o 256 MB
  - EEPROM/FLASH/FRAM: 0.5 kB, 8 kB, 64 kB, 256 kB, o 512 kB
- Arranque
  - Cartucho de NDS
  - Firmware FLASH
  - Wifi
  - Cartucho de GBA
- Fuente de alimentación
  - Batería de litio recargable integrada, 3.7V 1000mAh (DS-Lite)
  - Alimentación externa: 5.2V DC

La consola incorpora dos procesadores, un ARM9 a 66 MHz y un ARM7 a 33 MHz. En el primero se ejecuta la mayor parte del código, no siendo recomendable hacerlo en el ARM7. Se introdujo este procesador para mejorar el rendimiento 3D de la consola, aunque parece ser que, por motivos de arquitectura, su velocidad está desaprovechada. En el segundo se ejecutan tan solo ciertas funciones de la API. La razón de que posea dos procesadores cuando realmente solo se utiliza prácticamente uno es por compatibilidad con la GameBoyAdvance, la precursora de la Nintendo DS.

Como ya se comentó en el capítulo anterior, uno de los hándicaps de la videoconsola es su reducida memoria, tanto la principal como la de vídeo. La RAM principal es de 4096 kB. En esta memoria se cargan todas las entidades activas del código, tanto las variables como los modelos de la escena, sus texturas y el audio empleado. Un modelo relativamente complejo, con un número de polígonos entre 100 y 300, con información de materiales y animación puede ocupar entre 15 y 30 kB (si el modelo tiene muchos vértices, es decir más de 1000, puede ocupar más de 1000 kB). Las texturas, que son lo que más ocupa, y sus paletas pueden ocupar, dependiendo de sus dimensiones y la profundidad de bit de cada canal de color, hasta 64 kB. Esto significa que si no se tiene cuidado con el peso de los recursos se puede llenar rápidamente la memoria principal.

La VRAM es la memoria específica de vídeo. Ésta puede ser configurada de varias maneras, influyendo en el tipo de datos que se pueden almacenar en ella. Así, se pueden configurar en modos teselados (para mapas de tiles o teselas, con las que crear fondos de gran tamaño a partir de imágenes indexadas más reducidas, o para las propias teselas), para bitmaps (como fondos), para sprites o para texturas y sus paletas (tablas de colores indexados que se asignan en la imagen, de forma que se reduce el peso de ésta). Dispone de 656 kB y está repartida en 9 bancos de memoria (del A al I). Del A al D son de 128 kB, el E es de 68 kB, F y G son de 16 kB y H e I son de 32 kB y 16 kB respectivamente. Los dos últimos se emplean para las paletas, los demás para sprites y texturas. Cuando

se quiere utilizar una imagen es necesario cargarla previamente de la memoria principal a la memoria de vídeo. Por lo tanto, si se están utilizando por ejemplo varias texturas en una escena, correspondientes a varios modelos, éstas ocuparán un espacio en la VRAM. Si el número y el tamaño de las texturas son demasiado grandes, no se podrán almacenar todas en memoria y no se pintarán. Por esto es crucial controlar perfectamente la distribución de las texturas en dichos bancos, para además de no sobrepasar el límite total, no desaprovechar memoria en ninguno de ellos.

Para realizar el pintado en cada una de las pantallas, la Nintendo DS emplea dos motores de renderizado, el principal y uno secundario. De esta forma, a cada una de las pantallas se les asigna uno de los motores para pintar en ellas. Cada uno de estos motores tiene características diferentes, disponiendo el principal de más opciones y funcionalidades, además de tener acceso a más memoria de video. Cada motor tiene diferentes modos de funcionamiento, denominados modos de video, que especifican que tipo de gráficos se van a dibujar.

Así, disponen de varios modos 2D: el más simple es en el que se pinta una imagen de fondo por pantalla, con sprites superpuestos para representar personaje u objetos; en el segundo, más complejo, se disponen de hasta cuatro capas de fondo sobre las que pintar las imágenes de fondo (BG0, BG1, BG2, BG3), más una capa adicional para los sprites, superpuesta a las anteriores. De esta forma, se consigue el efecto de profundidad mediante imágenes planas. A las dos capas superiores (BG2 y BG3) se le pueden aplicar efectos de scroll y rotación.

El motor principal es el encargado del proceso de gráficos 3D, no teniendo el motor secundario esta funcionalidad, por lo que solo se pueden pintar entornos tridimensionales en una de las pantallas (aunque es posible, configurando adecuadamente el hardware, pintar 3D en ambas pantallas, con el inconveniente de reducir la fluidez). El pintado de los gráficos 3D se realiza en la capa de fondo BG0, por lo que, dependiendo del modo de pintado (2D o 3D), dicha capa se asocia a un motor u otro.

En la Figura 3.1 se muestra un diagrama del proceso de vídeo, mostrando el acceso a memoria de vídeo de los motores principal y secundario, la selección de las capas de fondo en función del motor empleado (2D o 3D) y la salida resultante a cada pantalla LCD.

3.2. Características del motor 3D

Se han especificado las características generales del hardware de la Nintendo DS. En este epígrafe se detallarán las características particulares del motor 3D de la consola, puesto que se trata del punto central de este proyecto. Se ahondará por lo tanto en su funcionamiento (el pipeline, es decir, la secuencia de operaciones que sigue) y en el proceso a seguir a la hora de mandar vértices, aplicar materiales, realizar transformaciones, etc. Este estudio es muy necesario para conocer las posibilidades del motor y así especificar los requerimientos y los objetivos en el desarrollo de la librería de renderizado implementada. Esto es así, puesto que, aunque se ha utilizado Libnds, la librería de soporte contenida en devkitPro con la que se accede a los recursos de la Nintendo DS, facilitando las tareas de configuración y comunicación con el hardware en el entorno 3D, si se pretende aprovechar al máximo las posibilidades del motor, es necesario pasar a una capa de abstracción inferior, comunicándose, en muchas ocasiones, directamente con ciertos

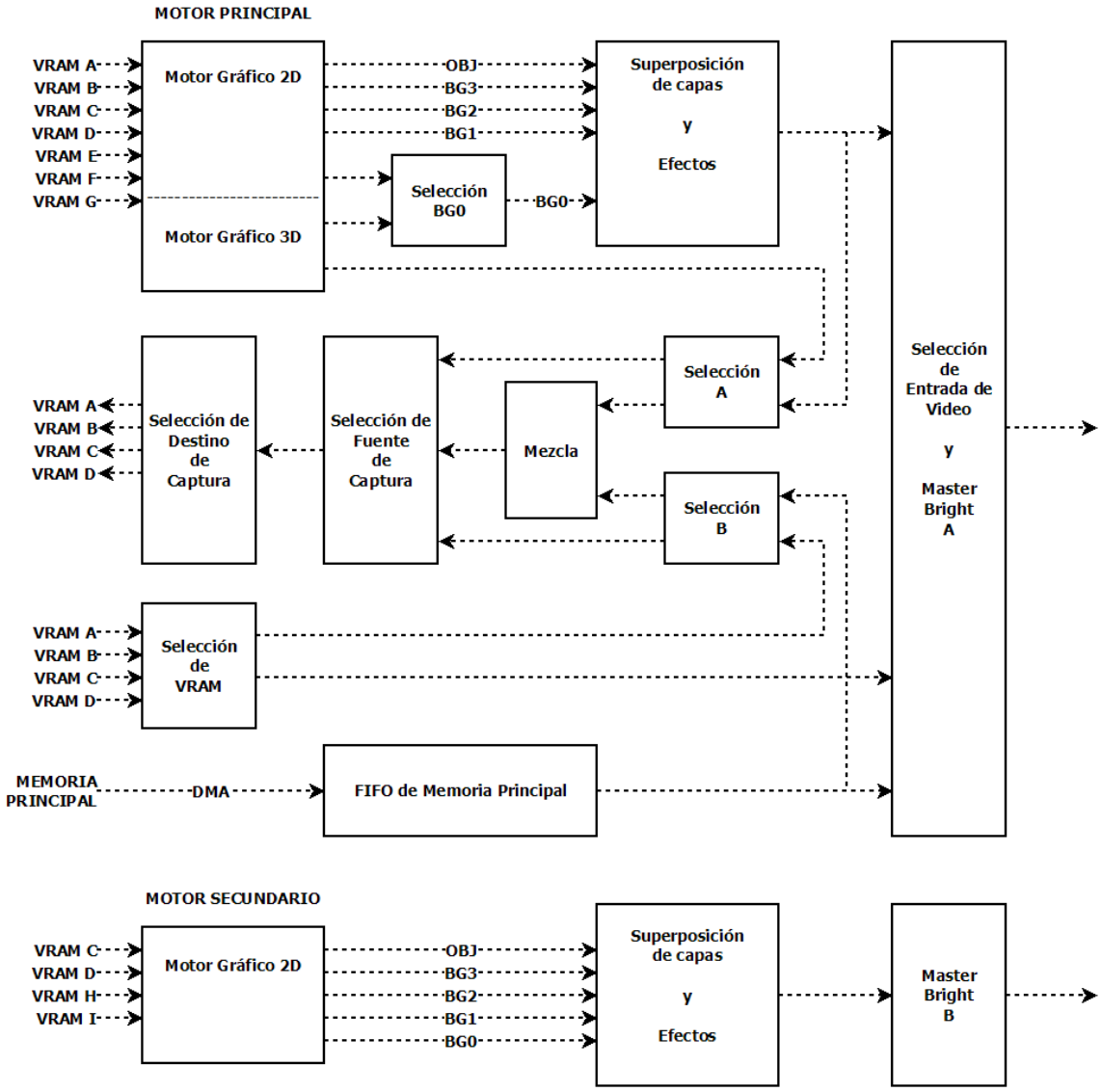

Figura 3.1: Diagrama del procesamiento de vídeo en la Nintendo DS

registros de entrada/salida del hardware.

3.2.1. Funcionamiento interno del chip gráfico de la Nintendo DS

Siguiendo el esquema comentado en el capítulo anterior, el chip gráfico de la Nintendo DS sigue una cadena de procesamiento (pipeline), que se divide en dos etapas: la de geometría y la de rasterizado. La etapa de aplicación es precisamente la librería de render, que dice qué vértices pintar, dónde y con qué características en cada momento, teniendo en cuenta la actualización de las entidades debido a las animaciones, la física, etc. De esta forma, al chip gráfico se le configura su estado, definiendo parámetros como la iluminación o las transformaciones, y se le entregan secuencialmente unas primitivas gráficas que éste se encargará de procesar atendiendo al estado del hardware y a los parámetros de dichas primitivas para ser representados inmediatamente según ciertas consignas.

Para entender mejor cómo funciona la Nintendo DS internamente y cómo interactúa con el programa, y en su extensión, con el usuario, se presenta a continuación el proceso que se ejecuta cuando se inicia una aplicación en la consola, y concretamente cuando ésta es una aplicación 3D.

- La CPU carga el programa en la memoria principal y éste se inicia.
- Lo primero que hace el hardware es atender a la configuración principal de sus sistemas, en este caso la inicialización y configuración del motor principal en modo 3D, el modo de video, la configuración de la memoria de video, etc.
- Una vez iniciados los motores (principal y secundario), éstos empiezan a trabajar de forma continua e ininterrumpida, de manera totalmente automática.
- El programa, una vez entra en el bucle principal, va cambiando registros y zonas de memoria en función de la información visual de la escena. Es decir, se manda la información relativa a los vértices, se determinan los parámetros de pintado y se se configura el estado del hardware.
- El hardware, trabajando de forma continua y automática, sigue la cadena de procesamiento fija, leyendo la memoria de video y los registros del sistema gráfico, finalizando en el pintado de la pantalla. Esto se realiza a razón de 60 ciclos por segundo.
- Por otra parte, la consola rellena ciertos registros automáticamente cuando se producen eventos a nivel físico. Éstos se denominan interrupciones, que el programa debe detectar y controlar, como puede ser la pulsación de los botones de la máquina.

De cara al programador, este proceso se puede definir más en detalle, sin entrar a comentar el código que hay que implementar, dado que este es un contexto de muy bajo nivel, en el que simplemente se tiene en cuenta el funcionamiento interno del hardware. Por lo tanto, a continuación se define un procedimiento genérico para el envío de geometría y configuración del hardware para visualizar modelos 3D en la Nintendo DS. Este proceso se explicará más en detalle en el capítulo referente a Libnds, donde se especifica el procedimiento y se detallan las funciones necesarias para ello.

- Lo primero es iniciar el hardware y configurar los parámetros de video necesarios para la normal visualización en las pantallas: definir la ubicación de la salida de los motores principal y secundario en cada una de las pantallas y definir el modo de video de cada uno (en este caso el motor principal trabajaría con el motor 3D, en el modo correspondiente).
- Configurar los estados del hardware que intervienen en el pintado:
  - iluminación: posición y configuración de hasta cuatro luces direccionales,
  - sombreado: la Nintendo DS dispone de varios tipos, como el toonshading o el sombreado Gouraud [referencia a wikipedia] estándar,
  - activación de anti-aliasing,
  - activación de la mezcla de transparencia: cómo se mezcla el color de un objeto translúcido con los objetos que tiene detrás,
  - contornos: se pueden definir varios colores de contornos para que dibujen las siluetas de los objetos,
  - niebla: crea un difuminado en base a la distancia del objeto a la cámara que

genera efecto de profundidad.

- Configurar la memoria de video: establecer qué bancos de memoria se van a emplear y para qué (texturas, paletas, fondos o sprites).
- Almacenar en memoria principal los recursos que se van a emplear (modelos y texturas).
- Definir las propiedades de los polígonos que se van a pintar: de qué forma les afecta la iluminación, su transparencia, si hay que pintar los polígonos que dan la espalda a la cámara, los que están de frente o ninguno (culling), si les afecta la niebla y definir su identificador (diferenciándolos entre sí para que funcionen el anti-aliasing y el contorno).
- Activar la textura que se quiere aplicar a estos polígonos (estableciendo la textura activa).
- Enviar los datos de geometría:
  - Especificar las primitivas que han de formar los polígonos: líneas, triángulos independientes, triángulos contiguos, quads (cuadriláteros) independientes y quads contiguos. De esta forma, el hardware sabe cómo procesar los vértices que se mandan a continuación y cómo unirlos entre sí.
  - Enviar las coordenadas de textura del vértice que se mandará a continuación
  - Enviar la normal del vértice que se mandará a continuación
  - Enviar los vértices.

Una vez introducido el procedimiento que hay que seguir para pintar geometría tridimensional en la Nintendo DS, es preciso describir de qué forma se llevan a cabo estos pasos y cómo se le manda al hardware todos estos datos.

Las etapas de geometría y rasterizado de la cadena de procesamiento, que efectúa automática y cíclicamente el chip gráfico, se pueden configurar (que no programar, ya que el proceso es fijo) accediendo a una serie de registros correspondientes a cada una. Así, se le dice al hardware con qué elementos debe trabajar y con qué parámetros, de forma que éste acceda a dichos registros y a la memoria de vídeo para realizar una tarea específica del pipeline.

3.2.2. Motor de geometría

Como ya se ha comentado, el motor de geometría realiza operaciones sobre vértices y polígonos. Por lo tanto, a través de los registros de memoria de los que dispone este motor se accede a los datos o comandos referentes a dichos elementos y se les pueden asignar atributos como el color del vértice, el número de vértices por polígono, realizar transformaciones sobre vértices, etc. De esta forma, si se quiere activar alguna propiedad, o mandar una lista de vértices, se hace por medio de los comandos correspondientes.

A continuación se pasa a enumerar los registros correspondientes al motor de geometría y a detallar la funcionalidad de los más importantes para el desarrollo de la librería de render.

Comandos de geometría

Para mandar comandos de geometría hay dos formas de hacerlo: por un lado, se puede mandar directamente un comando junto con su parámetro a la cola de comandos de geometría (FIFO). Éstos se pueden enviar de forma comprimida o descomprimida. En la primera, se envían al registro hasta cuatro comandos definidos por un id (8 bits cada uno) seguidos de los parámetros respectivos (32 bits cada uno). En la segunda, se envía el comando (los 24 últimos bits a 0) seguido de su parámetro.

Por otra parte, se pueden enviar los comandos a direcciones de registro específicas para cada uno. De esta forma, se manda el comando a su dirección particular, seguido de su parámetro. A continuación, comando y parámetros se envían al FIFO por parte de hardware.

Se envíen mediante su id o directamente a su registro, los comandos son los que se detallan a continuación. Éstos pueden agruparse según su funcionalidad. Es sumamente importante conocerlos, ya que se va a trabajar con ellos más adelante, en el desarrollo, si bien en muchas ocasiones se utilizarán las funciones que ofrece Libnds a tal efecto.

Uno de los grupos de **comandos de geometría** más importantes son los que intervienen en las operaciones con matrices. Éstas se suelen usar para realizar transformaciones sobre modelos, texturas, la propia cámara o cualquier tipo de operación matricial con matrices cargadas manualmente.

Un primer comando interviene en el **modo de matriz**. Hay cuatro modos de matriz: proyección, vista y modelo ("modelview"), posición y textura. Cada uno actúa sobre una pila de matrices. De esta forma, cuando se quiere operar sobre la matriz de transformación actual de cada modo, si se quiere conservar la anterior, ésta se almacena en la pila para poder recuperarla posteriormente.

Con el fin de realizar estas operaciones de inserción/recuperación de la pila se dispone de los siguientes comandos:

- **Insertar matriz en la pila (push)**: inserta la matriz actual en la pila correspondiente al modo de matriz.
- **Extraer matriz de la pila (pop)**: se extrae un número de matrices de la pila (definido por parámetro).
- **Almacenar matriz (store)**: se almacena la matriz en una dirección determinada (definida por parámetro).
- **Recuperar matriz (restore)**: se recupera la matriz almacenada en una dirección determinada (definida por parámetro).

Continuando con los comandos de geometría referentes al manejo de matrices se encuentran los que intervienen en las operaciones con éstas propiamente dichas. Éstas realizan procesos sobre la matriz actual. Estas operaciones son:

- **Cargar matriz identidad**: carga la matriz identidad en la matriz actual.
- **Cargar matriz de 4x4**: carga una matriz de dimensiones 4x4 en la matriz actual.
- **Cargar matriz de 4x3**: carga una matriz de dimensiones 4x3 en la matriz actual.
- **Multiplicar por matriz de 4x4**: multiplica la matriz actual por una matriz de 4x4.

- **Multiplicar por matriz de 4x3**: multiplica la matriz actual por una matriz de 4x3.
- **Multiplicar por matriz de 3x3**: multiplica la matriz actual por una matriz de 3x3.
- **Escalar**: multiplica la matriz actual por la matriz de escalado, con los parámetros correspondientes al escalado.
- **Trasladar**: multiplica la matriz actual por la matriz de traslación, con los parámetros correspondientes a la traslación.

Seguidamente se encuentran los **comandos que actúan sobre las propiedades de los vértices**. Con éstos se puede definir:

- el **color de los vértices** que se mandan a continuación: este es el color directo o color por vértice. Se aplica directamente al vértice, obviando cualquier tipo de propiedad material. Para establecer el color de un polígono se realiza una interpolación lineal entre los colores de los vértices que lo componen. Son 15 bits, 5 bits por componente rgb.
- las **normales** que correspondan a dichos vértices: 10 bits por componente (x, y, z) en punto fijo 1.9 (9 bits de parte fraccionaria, con signo).
- las **coordenadas de textura** del vértice: 16 bits por componente (u, v) en punto fijo 12.4 (4 bits de parte fraccionaria, con signo).
- las **coordenadas del vértice**: éstas se pueden enviar en diferentes formatos con más o menos precisión, o mandar únicamente dos componentes de las coordenadas en lugar de las tres:
  - formato **V10**: 10 bits por componente (x, y, z) en punto fijo 4.6 (6 bits de parte fraccionaria, con signo).
  - formato **V16**: 16 bits por componente (x, y, z) en 4.12 con signo. Para mandar los vértices en este formato hay que enviar dos veces el comando, la primera con las componentes x e y, la segunda con la componente z, de esta forma el hardware sabe que le estas mandando datos referentes a un vértice.
  - formato **XY**: se mandan únicamente las componentes (x, y), en formato 4.12.
  - formato **XZ**: se mandan únicamente las componentes (x, z), en formato 4.12.
  - formato **XY**: se mandan únicamente las componentes (y, z), en formato 4.12.

Estos comandos, exceptuando el de color de vértice, tienen que estar precedidos por el comando de inicio de envío de la lista de vértices y seguidos por el de finalización para que el chip gráfico sepa que le estas mandando información de vértices, aunque parece ser que el último tan solo es necesario por compatibilidad con OpenGL. Además, para una mejor eficiencia, es recomendable mandarlos según un orden: coordenadas de textura, normales y finalmente los vértices. El comando de vértice es obligatorio mandarlo el último para que se validen los anteriores.

La forma de enviar los comandos para pintar una malla es la siguiente: se recorren los triángulos de la malla; dentro de este bucle se recorren los vértices que componen cada triángulo; finalmente se envían los comandos de coordenadas de textura, de normal y de vértice, estableciendo así la información relativa al vértice del triángulo. De esta forma se hará para cada vértice del triángulo y para cada triángulo de la malla. Este sería el

proceso de forma esquemática:

COMANDO DE COLOR DE VÉRTICE (aplicado a todos los vértices enviados a continuación, en el caso de que éste sea el funcionamiento deseado)

COMANDO DE INICIO DE ENVÍO DE VÉRTICES

Recorrer triángulos de la malla

Recorrer vértices del triángulo

COMANDO DE COLOR DE VÉRTICE (en el caso de que se quiera aplicar únicamente al vértice considerado de forma independiente)

COMANDO DE COORDENADAS DE TEXTURA

COMANDO DE NORMAL

COMANDO DE VÉRTICE

Fin de recorrido de vértices del triángulo

Fin de recorrido de triángulos de la malla

COMANDO DE FINALIZACIÓN DE ENVÍO DE VÉRTICES

Con el siguiente comando, el de las **propiedades del polígono**, se definen las propiedades de los polígonos que se van a pintar. Entre estos atributos se encuentran:

- Las **luces** de la escena que han de afectar al polígono, cuyo efecto será calculado tras enviar el comando de normales.
- El **modo de mezcla** (*blending*) que ha de aplicársele. En este proceso se tienen en cuenta los colores que componen el color final del vértice (color, material, textura y transparencia) y se modulan de una determinada forma según el modo. Hay cuatro modos de mezcla: modulación (se mezclan todas las componentes de forma lineal), calcomanía, del inglés *decal* (parecido al anterior, solo que en este caso la transparencia de la textura sirve como coeficiente entre el color del vértice y el de la textura, de tal forma que el color que predomina es el de la textura en aquellas zonas donde el alfa de ésta no es alto), *toonshading* (igual que el modo modulación, solo que en vez de usar el color del vértice, calculado a partir del material y las normales, se extrae de una tabla de ponderación, consiguiendo una iluminación más plana, como de dibujos animados) y modo reflejos (igual que el *toonshading* solo que añadiendo un offset extra de intensidad) y finalmente el modo sombra (el polígono se pinta en negro). Los modos *toonshading* y el de reflejos van unidos en estas opciones y con otra opción del motor de renderizado se elige cuál de los dos usar en caso de estar esta opción activa.
- Un atributo que define si procesar o no las **caras traseras**.
- Un atributo que define si procesar o no las **caras delanteras**.
- Un atributo que define si almacenar o no la **profundidad de los píxels translúcidos**.
- Un atributo que define si pintar los **polígonos cortados por el plano lejano** (del *frustrum*) realizando el *clipping* o no pintarlos en absoluto.
- Un atributo que define si pintar o no **polígonos puntuales** (tan alejados que pueden ser pintados con un solo píxel).

- Operación a realizar tras el **test de profundidad**: o bien pintar los píxels con profundidad menor a los que ya hay almacenados, o bien pintar los que tengan la misma profundidad. Este parámetro es imprescindible para realizar render multi-pasada, como se explicará más adelante.
- Habilitar o no la **niebla** (*fog*) para este polígono.
- Asignación de la **transparencia** o **alfa** del polígono.
- Asignación del **identificador del polígono** (*polygon id*). Este atributo es necesario para que funcionen correctamente al anti-aliasing, la línea de contorno o las sombras volumétricas.

Los atributos que se asignen a este comando no se procesarán hasta que se envíe el comando de inicio de envío de la lista de vértices.

Continuando con los comandos del motor de geometría se encuentra el de **parámetros de textura**. Al igual que el de atributos del polígono, este comando asigna una serie de parámetros a la textura que se está procesando. Éstos son:

- **Localización** en la VRAM
- Parámetros que definen si hay que realizar **repetición de la textura**, tanto en el eje S (eje horizontal) como en el eje T (eje vertical).
- Parámetros que definen si hay que **realizar volteado** (flip) en la repetición de textura, tanto en el eje S como en el T.
- Parámetros que almacenan las **dimensiones** en S y en T de la textura en píxels.
- **Formato de la textura**. Puede ser:
  - Sin textura
  - A3I5 (3 bits canal alfa, 5 bits cada canal de color rgb)
  - 4 colores por paleta
  - 8 colores por paleta
  - 256 colores por paleta
  - Textura comprimida
  - A5I3 (5 bits canal alfa, 3 bits cada canal de color rgb)
  - Color directo
- Parámetro que define si hay que hacer el **primer color de la paleta de textura (color 0) transparente** o no. Esta transparencia no es gradual, sino que es total (alfa nulo). Esto es útil si se quiere prescindir de parte de la textura y conservar solo ciertas zonas de ésta (por ejemplo para hacer una valla, donde el espacio entre alambres está vacío).
- **Modo de transformación de las coordenadas de textura**. Estos pueden ser: sin transformación (la textura se mapea según las coordenadas de textura y no es posible transformarlas), por coordenadas de textura (aplica las coordenadas de textura y éstas pueden ser transformadas para conseguir efectos de movimiento), por normales (tiene en cuenta las normales para calcular las coordenadas de textura; necesario para el mapeo esférico) y por vértice o posición (mapeo de coordenadas de textura efectuado a partir de las coordenadas del vértice; para realizar movimientos

según las coordenadas de vista).

También referente a información de textura se encuentra el comando con el que se almacena la dirección de la **paleta de textura** correspondiente a la textura procesada.

Los siguientes comandos intervienen en los **materiales**. Por un lado, con el comando de color difuso/ambiente se definen simultáneamente estas componentes del material (15 bits cada componente, 5 por canal rgb, más un bit para indicar si se quiere usar el color difuso como color de vértice). De la misma forma, los colores especular y emisivo del material se definen mediante el comando de color especular/emisivo (15 bits cada componente, 5 por canal rgb, más un bit para indicar si se quiere usar una tabla predefinida para calcular la cantidad de brillo o *shininess*, definido mediante otro comando).

A continuación se encuentran los comandos que actúan sobre la **iluminación**. Por un lado está el comando con el que se define el vector de dirección de una luz determinada (definida también en este parámetro). Por otro lado está el que define el color de la luz seleccionada. Puede haber hasta 4 luces activas en la escena. Con estos parámetros, cuando se manda el comando de normales, el color del vértice se calcula ya no por el color directo que se haya especificado mediante el comando de color, si no teniendo en cuenta los parámetros de las luces que afectan al polígono (vector de dirección y color) y los de su material, según una serie de algoritmos encadenados (para más información sobre este proceso, ver sección "DS 3D Polygon Light Parameters", apartado "Internal Operationon Normal Command").

Un comando imprescindible para el correcto funcionamiento del sistema 3D es el de **intercambio de los *buffers***. Enviando este comando se pasan los *buffers* de polígonos y vértices actuales al motor de renderizado para ser pintados y los *buffers* que había en éste se pasan al motor de geometría para ser rellenados con más polígonos y vértices. Este comando está compuesto por dos parámetros de 1 bit cada uno: modo de manejo de polígonos translúcidos (automático o manual) y el modo de almacenamiento de la profundidad (mediante el valor Z o el valor transformado de éste W).

Con el comando de *viewport* se configura la **ventana de visualización** de la pantalla, que puede ser a pantalla completa o personalizada según los píxels de inicio y final en cada eje.

Finalmente, hay tres comandos para realizar diferentes test. El **box test** sirve para comprobar si un volumen cúbico se encuentra dentro del *frustrum*. Los otros dos, el **test de posición** y el de **vector de dirección**, se emplean para obtener dichos parámetros en las coordenadas de vista.

Otros registros del motor de geometría

Para finalizar de estudiar el motor de geometría, se comentarán el resto de registros que lo componen.

El registro correspondiente al estado del motor de geometría almacena información referente a éste como el estado de los test (libres u ocupados), el resultado del box test, el nivel en la pila de las matrices de posición y de vector, el de la matriz de proyección, el estado de la pila de matrices (ocupada realizando operaciones de inserción/extracción o no), información de errores en dicha pila, número de comandos enviados al FIFO, el estado del FIFO (si está lleno, a menos de la mitad de capacidad o vacío) o el estado del

motor de render (si está ocupado ejecutando comandos o no). Esta información es vital a la hora de ejecutar ciertos comandos, como por ejemplo los test.

El registro de contador de RAM contiene el número de polígonos y de vértices almacenados en la memoria destinada a la lista de polígonos y la de vértices respectivamente. Este dato puede resultar interesante para comprobar la eficiencia del renderizado y controlar que la geometría se mantenga dentro de unos límites.

Otro registro almacena la profundidad a partir de la cual los polígonos puntuales se pueden dejar de pintar, si esta opción está activada en el polígono. De esta forma se obvian cálculos innecesarios y se evitan imperfecciones visuales.

Por otro lado, en dos registros consecutivos se almacena el resultado de los test de posición y el de vector de dirección, previamente ejecutados y configurados mediante los comandos correspondientes.

Finalmente, los dos últimos registros correspondientes al motor de geometría almacenan las actuales matrices de clip y de vector direccional. La matriz de clip es el resultado de la multiplicación entre la matriz de posición y la de proyección.

3.2.3. Motor de rasterizado

Al igual que el motor de geometría, el motor de renderizado consta de una serie de registros a los que hay que acceder para configurar los aspectos del pintado de la escena. El área de acción de cada uno de estos registros se enumera a continuación:

- Control de la visualización 3D. Los parámetros de este registro se comentarán con más detalle más adelante, ya que tiene suma importancia en lo que se refiere a las opciones de render.
- Contador de número de líneas renderizadas
- Colores de los contornos (en caso de que la opción de pintarlos contornos esté activada). Pueden almacenarse hasta 8 colores.
- Referencia para el test de transparencia: almacena el valor del alfa por debajo del cual los objetos transparentes no han de pintarse.
- Color del plano del fondo (*rear plane* o *clear plane*).
- Profundidad del plano del fondo. Este valor suele ser el máximo para que el plano esté lo más al fondo posible.
- Offsets de movimiento de la imagen de fondo. Sobre el plano de fondo se puede poner una imagen, como un cielo con nubes por ejemplo. Pues bien, mediante este registro se pueden variar los parámetros de offset en x y en y para mover lateralmente esta imagen, para crear la ilusión por ejemplo de que las nubes se mueven.
- Color de la niebla que afecta a los polígonos.
- Offset de la niebla. Este parámetro controla el incremento de niebla en función de la profundidad.
- Tabla de densidad de la niebla. Se trata de una tabla que almacena 32 valores de densidad.
- Tabla de *toon*. Es la tabla con de la que se extrae el color del vértice que se usa en

los modos *toonshading* y el de reflejos. Éste se determina usando el valor del color rojo del vértice como índice en la tabla de *toon*. De esta forma, para un rango del valor del rojo en entrada, hay un único valor de salida correspondiente a cada uno de los canales rgb.

Control de la visualización 3D

Tal y como se anunció previamente, en este apartado se comentará con más detalle cada uno de los parámetros del registro de control de visualización 3D.

En este registro se pueden activar o desactivar parámetros de render muy concretos que afectan al acabado final del pintado de la escena. Dichos parámetros son:

- Mapeado de texturas. Con esta opción se activa o desactiva el proceso de texturizado.
- Sombreado del polígono. Si el modo de mezcla del polígono es *toon*/reflejos, con este parámetro se elige cuál de los dos implementar.
- Test de transparencia. Este parámetro activa o desactiva el test de alfa, según el cual, los píxels con un valor de alfa menor que el valor de transparencia de referencia dejan de pintarse.
- Mezcla de transparencia. Con este parámetro se habilita la mezcla de píxels con transparencia sobre píxels más profundos, permitiendo de esta forma el pintado de polígonos transparentes.
- Anti-aliasing. Con este parámetro se habilita o deshabilita el proceso de anti-aliasing. El llamado *aliasing* en gráficos por ordenador es un efecto que provoca variaciones bruscas en las siluetas de los objetos. Esta disfuncionalidad se puede reducir aumentando levemente la transparencia en los píxels que limitan la forma del objeto, de tal forma que se funde con el fondo.
- Contornos. Esta opción activa el pintado del contorno de aquellos polígonos que tengan el mismo identificador. Puede usarse tanto como efecto estético como señalizador de objetos.
- Modo de niebla. La niebla tiene parámetros de color y de transparencia. En el primero modo, se tienen en cuenta ambas, en el segundo tan solo la transparencia.
- Niebla. Parámetro que habilita o deshabilita la niebla sobre los polígonos a los que les afecte (configurable desde los atributos del polígono). Esta niebla difumina los colores en función de la distancia a la que estén de la cámara, para simular precisamente este efecto de profundidad.
- Variación de la niebla
- Indicación de carga insuficiente de líneas renderizadas en el buffer. Si el número de líneas decrece durante el render, esto significa que éste se vuelve más lento que la visualización a la salida. De esta forma, si este número decrece (activando éste parámetro), indica que hay que disminuir el número de polígonos.
- Sobrecarga de la memoria de polígonos y vértices. Si se sobrepasa el límite de polígonos o vértices que el motor puede manejar, éste parámetro se activa.
- Modo del plano de fondo. El plano de fondo puede ser un color opaco o un bitmap.

3.2.4. Conclusiones

Haciendo un estudio de las capacidades del motor 3D de la Nintendo DS se pueden establecer los límites del renderizador desarrollado en este proyecto, además de conocer el funcionamiento y la arquitectura del hardware gráfico de la consola, necesario para su desarrollo.

Recapitulando, el motor 3D de la consola tiene unas funcionalidades básicas poco programables, si bien en algunos casos son en cierta medida configurables.

A nivel de geometría, el motor es capaz de manejar, vértices, normales y coordenadas de textura. A éstos vértices se les puede asignar o bien un color directo, o bien unas propiedades materiales, tales como color difuso, ambiente, especular y emisivo. El nivel de brillo sin embargo se maneja simplemente mediante una tabla prefijada. Por lo tanto el proceso es el siguiente: asignar el material a cada vértice, asignar su textura, definir las propiedades del polígono, mandar el comando de inicio de lista de vértices, mandar las coordenadas de textura, mandar sus normales asociadas y finalmente enviar los vértices, cerrando el proceso con el comando de finalización de envío de vértices. Como se detallará más adelantes, este será efectivamente el proceso seguido en la función de render implementada.

La iluminación es sencilla, teniendo a disposición tan solo 4 luces en la escena. A cada polígono se le asigna qué luces le afectan. Para el cálculo de la iluminación sobre los polígonos basta con mandar el comando de normales. Para tal proceso, el motor usa las propiedades materiales asignadas a cada vértice, así como la textura.

Hay varios modos de mezcla, cuya utilización depende del efecto y la apariencia que se quiera generar. En la mayoría de los caso se empleará el de modulación, debido a que el resultado es más realista. Sin embargo, si se quiere dar la apariencia de que la textura está pegada sobre el objeto (pero es independiente, no forma parte de él), se empleará el modo pegatina o calcomanía (*decal*). Finalmente, si lo que se busca es una apariencia *cartoon*, de dibujos animados el modo empleado ha de ser el *toon*/reflejos.

En cuanto a las opciones referentes al motor de render, prácticamente la totalidad son activables pero no configurables (*alpha blending*, *alpha test*, anti-aliasing), aunque algunas sí lo son (la niebla y el plano de fondo).

Estas son por lo tanto las características que ha de manejar la librería, con el fin de sacar el máximo provecho al motor 3D de la Nintendo DS. Además, jugando con ciertas funcionalidades, se implementarán efectos y técnicas de render para mejorar el resultado final y sacarle más partido al motor, como el render multi-pasada o multi-capa, para aplicarle a un modelo varias texturas y mapeado esférico o el color por vértice, con el cual se puede prescindir del texturizado, ahorrando memoria de vídeo.

Capítulo 4

Librería Libnds

El objetivo de este proyecto es utilizar las capacidades gráficas de la Nintendo DS mediante la creación de una librería, donde se implementará un motor de render que saque partido de éstas permitiéndoles experimentar sobre sus capacidades.

Antes de pasar a la descripción y especificación de dicha librería, se describirá el proceso de desarrollo en la Nintendo DS. Se detallarán por lo tanto los requisitos y las herramientas necesarias, tanto de software como de hardware, para el desarrollo *homebrew* o casero de aplicaciones en la Nintendo DS.

Además, se ahondará en la API Libnds, sobre la cual se ha implementado dicha librería. Se comentarán por lo tanto las posibilidades de este soporte y más en particular de su módulo de video 3D, videoGL, ya que será determinante en el desarrollo.

4.1. Desarrollo en Nintendo DS

Como se ha comentado, la Nintendo DS se ha convertido, gracias al apoyo de la comunidad de desarrolladores *amateur* y a la proliferación de cartuchos destinados a realizar copias de seguridad, en una plataforma idónea para realizar proyectos caseros con recursos reducidos.

En este apartado se detallarán las herramientas necesarias, así como el proceso de configuración del entorno de desarrollo, para poder crear aplicaciones en la Nintendo DS sin necesidad de emplear los kits de desarrollo oficiales de Nintendo.

4.1.1. Homebrew

Homebrew se traduce en castellano como "hecho en casa" o simplemente casero. Son todas aquellas aplicaciones hechas por aficionados, sin ánimo de lucro, con herramientas *open source*, creadas también por la comunidad *amateur*. Aunque se hable de "aficionados", muchos de estos son verdaderos expertos en el campo de la informática. Sin embargo, el desarrollo de estas herramientas o aplicaciones no es su trabajo principal, haciéndolo únicamente como entretenimiento, o incluso aprendizaje, en su tiempo libre. Por esta razón, se necesita una gran comunidad que dé soporte y continúe el desarrollo, puesto que si no, las herramientas quedan desactualizadas y resultan inutilizables.

El aspecto legal es importante en este entorno, ya que se están empleando herramientas que facilitan los mecanismos de la piratería, siendo sencilla la carga de juegos "pirata" como por medio de las *flashcards*. Por ello, en el desarrollo *homebrew* hay que respetar dos requisitos: no realizar ninguna modificación a la consola y, por supuesto, no violar derechos de autor, ya sea publicando un juego propiedad de otra compañía, o bien, empleando recursos ajenos protegidos.

Para el desarrollo en Nintendo DS existen varias plataformas, que van desde librerías a motores de juego ya creados que permiten la creación de juegos con conocimientos limitados de programación.

La API más completa y extendida es Libnds. Fue creada por Michael Noland (Joat) y Jason Rogers (Dovoto). Actualmente, Dave Murphy (WinterMute) la mantiene y actualiza. Ésta otorga acceso a todas las funcionalidades de la consola, como el manejo de datos de entrada/salida, el Wifi, la manipulación de imágenes y sprites o el acceso a los motores, tanto los 2D como el 3D.

Sobre ésta se han creado algunas librerías que ejercen como capa intermedia, con el fin de simplificar el trabajo a los programadores menos experimentados, ya que el uso de Libnds puede ser algo complejo. Sin embargo, estas herramientas pueden resultar incompletas en según qué ámbitos y pueden sufrir errores graves ante sucesivas actualizaciones de Libnds, si éstas no se mantienen adecuadamente.

La librería más utilizada, y la que mejor refleja los problemas que acarrea la falta de mantenimiento, es PALib. En sus inicios, esta librería tuvo mucha difusión, ya que permitía hacer juegos de Nintendo DS de forma rápida y sencilla, puesto que ponía al alcance gran cantidad de utilidades sin necesidad de ser un experto de la programación. Sin embargo, una de las principales desventajas de PALib frente a desarrollar en Libnds, aunque ésta sea más compleja y de más bajo nivel, es que no cuenta con soporte para gráficos 3D, ya que se limita al uso de los motores 2D.

Sin embargo, la falta de mantenimiento ha provocado que esta librería genere actualmente multitud de errores, haciendo el desarrollo prácticamente imposible. Por esta razón, PALib ha quedado fuera del panorama homebrew, hasta tal punto que en la página web de DevkitPro [8] avisan de las dificultades de desarrollar con PALib y que no se prestará ayuda a quienes pasen por alto esta advertencia. En 2012 la página oficial de la librería fue desmantelada, por lo que la propia librería ha sido oficialmente abandonada.

Otra librería interesante es NFLib [15], creada por Cesar Rincón (NightFox). Como PALib, esta librería actúa de capa intermedia entre Libnds y el desarrollador, facilitándole en gran medida la tarea. Sin embargo, ésta también se centra en los gráficos 2D, sin acceso a las funcionalidades del motor 3D. No obstante, se implementó una extensión, que no ha pasado de prototipo, en la que se ofrece acceso a las capacidades 3D de la Nintendo DS, siempre a través de Libnds. Sin embargo, las posibilidades que ésta ofrece son escasas (pintar triángulos y quads con texturas) y su última actualización data de 2011, por lo que parece que ha sido dejada de lado, de momento al menos. En lo que respecta a la propia NFLib, ésta sigue manteniéndose con actualizaciones paralelas a las de Libnds.

Por otro lado, una librería muy interesante es Nitro Engine [17], desarrollada por Antonio Niño, y también basada en Libnds. Ésta sí que está enfocada al 3D, ofreciendo métodos para cargar modelos, instanciar copias de éstos, animarlos (con o sin interpolación lineal para suavizar la animación), gestionar y aplicar texturas, efectos de partículas, etc. Además, incorpora física de colisión básica, funciones de API, captura de pantalla y

otras utilidades que convierten a Nitro Engine en lo que su nombre indica: un motor de juego, ya que con esta herramienta basta para crear la aplicación al completo. Una de las funcionalidades estrella es el "3D dual", es decir, la utilización "simultánea" de las dos pantallas para visualizar 3D, interesante ya que por hardware tan solo se puede asignar una de las dos pantallas al motor 3D. Esto lo debe conseguir alternando la asignación de la pantalla al motor 3D en cada frame (junto con el pintado del contenido de cada una), por lo que la tasa de frames pasa de 60 fps a 30 fps (en cada una de las pantallas).

Dadas estas funcionalidades, Nitro Engine es una buena herramienta para desarrollar juegos en 3D, puesto que simplifica sustancialmente el manejo de datos (mallas y texturas) y el pintado de polígonos, texturizado, etc. Sin embargo, la última actualización data de Septiembre de 2011, por lo que, tras las sucesivas actualizaciones de Libnds, la librería puede ser inestable.

Por lo tanto, debido a la notoriedad (totalmente fundada) que ha adquirido Libnds, las utilidades que aporta en lo referente a los gráficos 3D y el empeño de sus creadores de mantenerla viva, se ha decidido emplear esta librería para desarrollar el motor de render con el que se evaluarán las capacidades gráficas de la Nintendo DS.

4.1.2. Entorno de desarrollo

En este epígrafe se detallarán las herramientas de software y de hardware utilizados en el desarrollo de la librería, así como su configuración y puesta en marcha.

Hardware

El desarrollo en Nintendo DS se realiza en PC, dónde se genera el ejecutable con el que trabaja la consola. En este caso se ha utilizado el sistema operativo Windows (más concretamente Windows 7), aunque también es posible desarrollar en Linux.

Con el fin de probar los resultados que se van obteniendo es necesario utilizar una Nintendo DS como máquina de pruebas. Aunque es posible utilizar emuladores para cargar ejecutables de Nintendo DS en el ordenador, éstos no siempre se comportan exactamente igual que la consola (hay ciertas funcionalidades que en según qué emuladores no se simulan correctamente, particularmente algunos detalles de los gráficos 3D), por lo que es recomendable realizar pruebas en la propia máquina. En este caso, se ha utilizado una Nintendo DS Lite.

Para cargar dichos archivos en la consola, se emplean los denominados *flashcards*. Éstos son cartuchos de Nintendo DS, a los que se les introduce una memoria flash mini SD. De esta forma, con un lector de tarjetas mini SD (en este caso se ha empleado un adaptador USB), se pasan los ejecutables y los archivos de configuración y guardado de una plataforma a otra. Además, incorporan un firmware con el que acceder a diferentes funcionalidades dependiendo del tipo de tarjeta. Éstas pueden ser simplemente acceder a los archivos almacenados, reproducir videos o música, agenda electrónica, etc.

Hay una gran variedad de marcas de *flashcards* en el mercado y cada una tiene sus propias características, si bien la función básica de todas ellas es el almacenamiento y ejecución de archivos de Nintendo DS (extensión .nds). En este proyecto se ha utilizado una M3D Real con una tarjeta mini SD de 2 GB.

Software

Es necesario instalar y configurar ciertos programas y librerías para poder desarrollar aplicaciones de Nintendo DS.

Es primordial y obligatorio instalar devkitPro, un conjunto de herramientas de desarrollo para Wii (Nintendo), GameCube (Nintendo), Nintendo DS (Nintendo), GameBoyAdvance (Nintendo), GamePark 32 (Game Park) y Playstation Portable (Sony). En este caso se instalará tan solo el módulo de Nintendo DS, que consta de la librería Libnds, algunas herramientas y una buena selección de ejemplos de utilización.

Para ello basta con ir a la página oficial [8], y descargar la última versión. Al ejecutar el archivo descargado se inicia un asistente de instalación. Éste permite especificar la carpeta de instalación y los módulos que se desean instalar (en este caso tan solo es necesario instalar el correspondiente a Nintendo DS).

Una vez instalado devkitPro, ya se tienen las librerías C/C++ necesarias para desarrollar en PC. Para la edición de código hay varias opciones. Un editor de código fuente bastante extendido entre la comunidad es el Porgrammer's Notepad, ya que viene incluido en el devkitPro. Sin embargo, en este proyecto se ha utilizado el entorno de desarrollo integrado (IDE, según las siglas en inglés) Microsoft Visual Studio Express 2008. Es una versión gratuita del Visual Studio un poco más limitada pero para este desarrollo perfectamente funcional. Se ha optado por éste, en vez de por el Porgrammer's Notepad, debido principalmente a que es un poco más avanzado en cuanto a la edición de código fuente (permite acceder a los diferentes archivos incluidos en el proyecto de forma rápida, etc) y además se le puede añadir un *plugin* para crear proyectos específicos de Nintendo DS, como se explicará más adelante. La instalación del Visual Studio Express 2008 es común a la de cualquier programa: se descarga el programa de la página web de Microsoft y se instala siguiendo las instrucciones del asistente de instalación.

En estos momentos puede ser complicado encontrar esta edición en la página de Microsoft ya que es relativamente antigua y ya existen dos versiones posteriores a ésta (la 2010 y la 2012).

Como se ha comentado, existe un *plugin* libre que permite crear proyectos de devkitPro (Nintendo DS/Game Boy Advance, Wii y GameCube). El programa en cuestión se llama Nintendo DS Wizard y fue creado por Jason Rogers (Dovoto). Tiene que estar instalado en la carpeta de instalación de Visual Studio 2008.Éste incorpora una plantilla que genera un archivo C con las inclusiones básicas (libnds y la librería estándar de entrada/salida), así como una función principal donde se inicia la pantalla de la consola, se imprime en pantalla "Hola mundo" y se inicia el bucle principal. Además, incluye el archivo MAKE que se utilizará para generar el ejecutable.

El proceso de instalación y configuración de todas estas herramientas viene explicado y detallado mediante videos en la página de tutoriales de NightFox [16].

Una vez hecho el proceso de instalación y configuración del software necesario, se puede comenzar el desarrollo de aplicaciones para la Nintendo DS.

Para probar las aplicaciones implementadas, es recomendable emplear un emulador, que simula el proceso interno de la consola en el ordenador sobre OpenGL, con el fin de no tener que pasar continuamente el programa a la Nintendo DS, agilizando así el proceso de desarrollo/pruebas. Sin embargo, los emuladores no trabajan al cien por cien de la misma

forma que la consola real, no siendo simuladas ciertas funcionalidades correctamente. Por ello, siempre es necesario realizar algunas pruebas en la consola, para comprobar si es la aplicación la que falla o bien el emulador el que no responde correctamente.

Hay varios emuladores en la escena *homebrew*, cada uno presentando sus defectos y virtudes. Los más extendidos son DeSmuME [7], iDeaS [12] y NO$GBA [18]. De entre estos, el que parece ser más potente es el último, simulando especialmente bien el proceso 3D.

4.2. Libnds

Dado que se va a emplear la librería Libnds como soporte al motor de render desarrollado en este proyecto, es necesario conocer a fondo las especificaciones y características de ésta.

Por lo tanto, en este epígrafe se detallarán algunos de los módulos básicos de la librería esencialmente los más importantes y los que se han empleado específicamente en el desarrollo. Éstos son el de input, el de sistema de archivos, el de vídeo y el de la consola de comandos. Hay muchos otros que permiten el manejo de memoria, el sonido, el tiempo, los sprites, los fondos, etc. Sin embargo, a éstos últimos no se accede de forma explícita en la librería de render (sí de forma implícita a través de otros módulos), por lo que, para no alargar en exceso la enumeración, se obviarán en este texto. Si se desea obtener más información sobre todos los módulos de la librería se recomienda acceder a la documentación de Libnds [13].

Más tarde, se detallará de forma especial el de gráficos 3D, una versión simplificada de OpenGL para Nintendo DS, que ha de conocerse a fondo para crear aplicaciones 3D.

4.2.1. Input.h

El módulo de *input* es el que da acceso a los datos de entrada desde el interfaz de control de la consola (botones y pantalla táctil). Con las funciones que proporciona, se puede obtener la pulsación de los botones, los botones que se están manteniendo pulsados y la posición de contacto de la pantalla táctil. El proceso de obtención del *input* en cada ciclo es el que se detalla a continuación.

Una vez por frame (y solo una, si no puede acarrear fallos), es decir, en el bucle principal, se llama a la función *scanKeys*. Ésta activa la lectura del *input*, obteniéndose internamente los estados actuales del pad de control. Por otra parte, hay diferentes funciones que devuelven dichos estados, almacenados en enteros sin signo (32 bits), de los cuales cada uno de los bits representa el estado de cada botón. De esta forma, se comprueba si un botón determinado está en dicho estado en ese instante de tiempo o ciclo. Esto se hace mediante máscaras de bits. Así, si en la palabra de 32 bits que almacena el estado del pad el bit correspondiente a un botón determinado está activado, significará que dicho botón cumple este estado. Estas funciones y sus estados correspondientes son:

- **keysCurrent**: devuelve el estado actual del pad de control.
- **keysDown**: devuelve el estado actual de pulsación del pad de control (se pulsa un botón).
- **keysDownRepeat**: devuelve el estado actual de pulsación o repetición del pad de

control (se pulsa un botón o se repite la pulsación).

- **keysHeld**: devuelve el estado actual de pulsación mantenida del pad de control (se mantiene pulsado un botón).
- **keysSetRepeat**: configura los parámetros de repetición.
- **keysUp**: devuelve el estado actual de liberación del pad de control (se librea la pulsación de un botón).
- **touchRead**: devuelve el estado actual del panel táctil.

Así, almacenando dichos estados en variables y contrastando éstas con los diferentes botones del pad, se controla la información de entrada del pad de control de la Nintendo DS. Los valores asignados a cada uno de los botones están definidos en este módulo como elementos del enumerador KEYPAD_BITS.

4.2.2. System.h

Este módulo incorpora utilidades de acceso al hardware de la Nintendo DS. En este proyecto, principalmente se han empleado las funciones que definen qué pantalla ha de estar asociada al motor principal (el que tiene soporte 3D). Éstas son: *lcdMainOnBottom* (fuerza la visualización del motor principal en la pantalla inferior); *lcdMainOnTop* (fuerza la visualización del motor principal en la pantalla superior); *lcdSwap* (intercambia la visualización de las pantallas).

4.2.3. Video.h

A través del módulo de vídeo se acceden a las funcionalidades generales de vídeo. Éstas son comunes para los motores de 2D y 3D. Es a través de este módulo que se accede a la configuración de los bancos de memoria de vídeo, entre otras funcionalidades.

Así pues, en el módulo de vídeo se definen la dirección que apunta a los distintos bancos de memoria. De esta forma, se puede configurar el modo de cada uno de éstos mediante las funciones *vramSetBankX* (VRAM_X_TYPE x), donde X especifica el banco de VRAM (del A al I) y x es el tipo que se le quiere asignar a dicho banco. Estos tipos están de igual manera definidos para cada uno de los bancos, según los tipos que admita cada uno de ellos. Así, se pueden configurar los bancos del A al D para fondos, *sprites* o texturas y los bancos del E al I para fondos, *sprites* y paletas de textura.

A través del módulo de video también se puede configurar el modo de vídeo que se va a utilizar en cada uno de los motores (el principal y el secundario). De esta forma se puede configurar el motor principal para que sea 2D o 3D y según el modo en cada caso, se podrán utilizar los fondos disponibles (hasta 4) para realizar acciones específicas (pintar texto, rotaciones o definir el fondo como un bitmap de gran tamaño).

Además, video.h incorpora una función encargada de activar el 3D, *video3DEnabled* (), a la que hay que llamar al inicio de la aplicación para poder trabajar con el motor 3D.

El módulo de video incorpora, además, funciones para activar y desactivar fondos en los motores principal (*videoBgEnable* y *videoBgDisable*) y secundario (*videoBgEnableSub* y *videoBgDisableSub*), así como una función para ajustar el brillo de las pantallas (*setBrightness*).

Por último, mencionar que incorpora una macro considerablemente útil que convierte valores separados de color r, g y b en un triplete con formato de color específico de la Nintendo DS RGB o ARGB en caso de añadirle la transparencia.

4.2.4. Console.h

Éste módulo aporta funcionalidades para la fase de depuración, en cuanto a que desde éste se puede activar el modo de consola, en el que se pueden imprimir texto utilizando la librería estándar de entrada/salida.

En este proyecto, y aunque la librería pone a disposición más funciones, se he empleado básicamente la que inicializa la consola en modo prototipo (*consoleDemoInit*), en el que se activa para ello el fondo 0 (BG0) en modo texto, permitiendo la impresión.

Además de para depurar, y como se verá en el capítulo correspondiente al desarrollo de la aplicación de demostración, este método de impresión de texto es útil para generar menús y textos indicativos según el contexto del programa.

4.2.5. Trig_lut.h

Éste módulo aporta funciones trigonométricas básicas de punto fijo, tales como el seno, el coseno, la tangente, el arcoseno y el arcocoseno.

Además dispone de conversores que transforman enteros o flotantes a punto fijo. Ésta utilidad es considerablemente útil, teniendo en cuenta que la Nintendo DS trabaja en este formato y es recomendable manejar y emplear los datos en dicho formato que manejar flotantes y que Libnds, por debajo, tenga que realizar las conversiones. Ésta es la recomendación que hacen en la documentación de la librería.

Se pueden realizar conversiones de entero a punto fijo (*intToFixed*), de flotante a punto fijo (*floatToFixed*) y viceversa (*fixedToInt* y *fixedToFloat* respectivamente). Además se pueden realizar conversiones de ángulo de grados a radianes (*degreesToAngle*) y viceversa (*angleToDegrees*).

4.3. API 3D: videoGL

La librería Libnds pone a disposición del usuario una API gráfica con la que acceder a las características 3D de la Nintendo DS. Ésta está contenida en el módulo de vídeo específico para gráficos, videoGL. Dicha API gráfica está concebida como una versión reducida de OpenGL para la Nintendo DS. Esto resulta comprensible ya que ambas plataformas trabajan de forma similar, en tanto que funcionan como máquinas de estado a las cuales se les van enviando comandos que ejecutar (esto es cierto para las versiones de OpenGL anteriores a la 3.2).

De esta forma, las funciones que se encuentran en este interfaz son análogas a las que se emplean en OpenGL, teniendo en cuenta que internamente no realizan las mismas tareas, puesto que videoGL de Libnds está concebida específicamente para el chip gráfico de la Nintendo DS.

Por otra parte, el abanico de opciones que presenta esta API con respecto a OpenGL está mucho más limitado, ya que las capacidades técnicas de la consola están igualmente limitadas.

Dado que se va a emplear esta librería para realizar el motor de render, es necesario hacer un estudio de las posibilidades y las opciones que ofrece su API gráfica. Por lo tanto, en este apartado se detallarán las funciones y las utilidades que componen videoGL, para más tarde poder aplicarlas en el desarrollo de la herramienta.

4.3.1. Funciones

En este apartado se especificarán las funciones definidas en el módulo videoGL, con el fin de familiarizarse con éstas y comprender así como se opera en el modo de gráficos 3D de la Nintendo DS.

Matrices

La librería pone a disposición una serie de funciones para el manejo y las operaciones con matrices. Estas funciones, internamente, mandan al hardware los comandos de geometría correspondientes con los parámetros deseados.

La Nintendo DS trabaja con el formato f32, es decir, números de punto fijo con signo de 32 bits, con una parte fraccionaria de 12 bits. En algunos casos existen diferentes versiones para una misma función, de forma que se puedan manejar datos tanto en f32 como en punto flotante. Sin embargo, en éste último caso, internamente se han de convertir los datos de entrada a f32. Por esta razón, es recomendable, y así lo especifica la documentación de Libnds, pasar los datos directamente en formato f32.

En primer lugar se encuentran las funciones que intervienen en el manejo de las matrices de cada pila (proyección, modelo, posición y textura).

- void glMatrixMode (GL_MATRIX_MODE_ENUM mode): define la matriz con la que se va a operar a continuación.
- void glPopMatrix (intnum): envía el comando de extracción de matrices, con el que se extraen el número especificado (num) de matrices de la pila.
- void glPushMatrix (void): envía el comando de inserción de matriz, con el cual se inserta la matriz actual en la pila.
- void glResetMatrixStack (void): devuelve todas las matrices al nivel superior de la pila.
- void glRestoreMatrix (intindex): recupera una matriz de la pila, seleccionada con index, y la asigna a la matriz actual.
- void glStoreMatrix (intindex): almacena la matriz actual en una posición de la pila, determinada por index.
- void gluPickMatrix (int x, int y, intwidth, intheight, constintviewport[4]): genera una matriz de selección, útil para determinar objetos seleccionados en el panel táctil.

Por otro lado, se encuentran las funciones para operar con matrices, ya sea para multiplicar la matriz actual por una matriz definida por el usuario, o bien por las matrices

de traslación, rotación y escalado. Además, en esta categoría se engloban también las funciones para operar sobre la matriz de proyección.

Funciones para cargar matrices:

- void glLoadIdentity (void): carga la matriz identidad en la matriz actual.
- void glLoadMatrix4x3 (const m4x3 *m): carga una matriz de 4x3 en la matriz actual.
- void glLoadMatrix4x4 (const m4x4 *m): carga una matriz de 4x4 en la matriz actual.

Funciones de multiplicación de matrices:

- void glMultMatrix3x3 (const m3x3 *m): multiplica la matriz actual por una matriz de 3x3.
- void glMultMatrix4x3 (const m4x3 *m): multiplica la matriz actual por una matriz de 4x3.
- void glMultMatrix4x4 (const m4x4 *m): multiplica la matriz actual por una matriz de 4x4.

Funciones para operar sobre la matriz de proyección:

- void glOrtho (float left, float right, float bottom, float top, float zNear, float zFar): multiplica la matriz actual en modo ortogonal. Versión de punto flotante.
- void glOrthof32 (int left, int right, int bottom, int top, int zNear, int zFar): multiplica la matriz actual en modo ortogonal. Versión f32.
- void gluPerspective (float fovy, float aspect, float zNear, float zFar): establece la matriz de proyección en modo perspectiva. Versión de punto flotante.
- void gluPerspectivef32 (int fovy, int aspect, int zNear, int zFar): establece la matriz de proyección en modo perspectiva. Versión f32.

Funciones para multiplicar por la matriz de traslación:

- void glTranslatef (float x, float y, float z): multiplica la matriz actual por una matriz de transformación. Versión de punto flotante.
- void glTranslatef32 (int x, int y, int z): multiplica la matriz actual por una matriz de transformación. Versión f32.
- void glTranslatev (const GLvector *v): multiplica la matriz actual por una matriz de transformación. Versión que soporta un vector de enteros (x, y, z).

Funciones para multiplicar por la matriz de rotación:

- void glRotatef (float angle, float x, float y, float z): realiza una rotación, definida por el ángulo de rotación (angle) y los ejes de aplicación (x, y, z). Versión de punto flotante.
- void glRotatef32 (float angle, int x, int y, int z): realiza una rotación, definida por el ángulo de rotación (angle) y los ejes de aplicación (x, y, z). Versión f32.
- void glRotatef32i (int angle, int x, int y, int z): realiza una rotación, definida por el ángulo de rotación (angle) y los ejes de aplicación (x, y, z). Versión de enteros.
- void glRotateX (float angle): realiza una rotación alrededor del eje x. Versión de

punto flotante.

- void glRotateXi (int angle): realiza una rotación alrededor del eje x. Versión de enteros.
- void glRotateY (float angle): realiza una rotación alrededor del eje y. Versión de punto flotante.
- void glRotateYi (int angle): realiza una rotación alrededor del eje y. Versión de enteros.
- void glRotateZ (float angle): realiza una rotación alrededor del eje z. Versión de punto flotante.
- void glRotateZi (int angle): realiza una rotación alrededor del eje z. Versión de enteros.

Funciones para multiplicar por la matriz de escalado:

- void glScalef (float x, float y, float z): multiplica la matriz actual por una matriz de escalado. Versión de punto flotante.
- void glScalef32 (int x, int y, int z): multiplica la matriz actual por una matriz de escalado. Versión f32.
- void glScalev (const GLvector *v): multiplica la matriz actual por una matriz de escalado. Versión que soporta un vector de enteros (x, y, z).

Vértices

En este apartado se especifican las funciones de videoGL cuyo objetivo es mandar los comandos de vértices, así como sus atributos, tales como sus normales, sus coordenadas de textura o sus propiedades materiales.

- void glBegin (GL_GLBEGIN_ENUM mode): envía el comando de inicio de lista de vértices. Es necesario llamar a esta función antes de enviar información relativa a los vértices. Ésta establece las primitivas que se van a construir con los vértices que se envíen a continuación. Éstas son: líneas, triángulos independientes, triángulos contiguos (se genera un nuevo polígono a partir del último vértice de la primitiva anterior), cuadriláteros independientes y cuadriláteros contiguos.
- void glEnd (void): envía el comando de finalización de lista de vértices. Este comando parece no afectar realmente al funcionamiento del motor 3D, pudiendo existir por compatibilidad con OpenGL. Según la especificación, esta función puede ser obviada.
- void glVertex3f (float x, float y, float z): envía el comando de vértice, especificando la posición de éste en coordenadas locales. Versión de punto flotante. Se recomienda usar la versión de v16 (número de punto fijo de 16 bits, con signo y 12 bits de parte fraccionaria), uno de los dos formatos con los que la Nintendo DS maneja los vértices (el otro es v10 (número de punto fijo de 10 bits, con signo y 6 bits de parte fraccionaria), aunque en la librería no se contempla su utilización).
- void glVertex3v16 (v16 x, v16 y, v16 z): envía el comando de vértice, especificando la posición de éste en coordenadas locales. Versión v16.
- void glColor (rgb color): envía el comando de color, con el color especificado en

formato rgb (16 bits, 5 por componente más uno de alfa, que en esta función es ignorado), que define directamente el color empleado a continuación.

- void glColor3b (uint8 red, uint8 green, uint8 blue): envía el comando de color, con el color especificado en 3 parámetros en formato uint8 (8 bits, cuyos 3 últimos son ignorados), que define directamente el color empleado a continuación.
- void glColor3f (float r, float g, float b): envía el comando de color, con el color especificado en 3 parámetros en formato de punto flotante (convertidos), que define directamente el color empleado a continuación.
- void glMaterialf (GL_MATERIALS_ENUM mode, rgb color): envía el comando de material especificado por mode (), con el que se asigna el color de dicha componente del material.
- void glMaterialShinyness (void): genera una tabla para la amplitud del resalte especular, que utiliza la Nintendo DS para aplicar dicho parámetro.
- void glNormal (u32 normal): envía el comando normal, que especifica la normal de los siguientes vértices. Versión que maneja normales empaquetadas (3 parámetros en v10 empaquetados en una palabra de 32 bits).
- void glNormal3f (float x, float y, float z): envía el comando normal, que especifica la normal de los siguientes vértices. Versión que maneja 3 parámetros en formato de punto flotante. Se recomienda usar la versión de normales empaquetadas.
- void glTexCoord2f (float s, float t): envía el comando de coordenadas de textura, que especifica las coordenadas de textura de los siguientes vértices. Versión de punto flotante. Se recomienda usar la versión t16.
- void glTexCoord2f32 (int u, int v): envía el comando de coordenadas de textura, que especifica las coordenadas de textura de los siguientes vértices. Versión de enteros.
- void glTexCoord2t16 (t16 u, t16 v): envía el comando de coordenadas de textura, que especifica las coordenadas de textura de los siguientes vértices. Versión t16 (16 bits con signo y 4 bits de parte fraccionaria).

Polígonos

A continuación se especifican las funciones que intervienen en la aplicación de propiedades a los polígonos.

- void glPolyFmt (u32 params): aplica los atributos de los polígonos que se pintan a continuación. Éstos se pasan realizando la suma binaria de los valores de cada parámetro contenido en GL_POLY_FORMAT_ENUM que se quiera aplicar, activando así el bit correspondiente del registro.
- u32 POLY_ALPHA (int n): establece la transparencia del polígono. El rango es de 1 (totalmente translúcido) a 31 (opaco). El valor 0 manda pintar los polígonos en modo alambre (wireframe).
- u32 POLY_ID (int n): establece el identificador del polígono. Hay 64 valores posibles. Se emplean identificadores distintos para que funcionen la transparencia, el antialiasing y el contorno.

Texturas

En este apartado se detallan las funciones de generación y manejo de texturas.

En algunos casos, se establece como parámetro de entrada a las funciones un entero target. Éste no se emplea para nada, existiendo únicamente por analogía con OpenGL, por lo que normalmente se deja a 0.

- int glGenTextures (int n, int *names): genera la lista de nombres de las texturas que se van a emplear.
- int glDeleteTextures (int n, int *names): destruye la lista de nombres previamente generada.
- void glBindTexture (int target, int name): define la textura definida por name como la textura activa.
- int glTexImage2D (int target, int empty1, GL_TEXTURE_TYPE_ENUM type, int sizeX, int sizeY, int empty2, int param, const void *texture):
  carga una textura en memoria (VRAM) y la asigna a la textura activa, estableciendo sus atributos (tipo de textura, dimensiones, parámetros definidos por la suma binaria de los elementos de GL_TEXTURE_PARAM_ENUM que se deseen aplicar y los datos de la propia textura cargados en memoria principal).
- void glTexParameter (int target, int param): establece lo parámetros de la textura activa, mediante la suma binaria de los elementos de GL_TEXTURE_PARAM_ENUM que se deseen aplicar.
- void glColorTableEXT (int target, int empty1, uint16 width, int empty2, int empty3, const uint16 *table): carga una paleta en memoria (VRAM) y la establece como paleta de la textura activa.
- void glAssignColorTable (int target, int name): establece como paleta de la textura activa una paleta de otra textura previamente cargada.
- u32 glGetTexParameter (void): devuelve los parámetros de la textura activa.
- void * glGetTexturePointer (int name): devuelve la dirección de memoria de la textura especificada por name.
- void glResetTextures (void): destruye todas las texturas en uso, liberando su espacio en memoria.

Configuración 3D

En este epígrafe se especifican las funciones relativas a la puesta en marcha y configuración del motor 3D de la Nintendo DS. A través de ellas se inicia el sistema 3D, se define la ventana de visualización y se activan y configuran distintos procesos.

En primer lugar se encuentran las funciones del sistema.

- void glInit (): inicializa la máquina de estados. Esta función hay que llamarla una vez, al principio del programa, para poder llamar al resto de funciones gl.
- void glEnable (int bits): activa los procesos definidos por la máscara de bits determinada a partir de los atributos de DISP3DCNT_ENUM (mezcla, anti-aliasing, test de transparencia, etc).

- void glDisable (int bits): desactiva los procesos definidos por la máscara de bits determinada a partir de los atributos de DISP3DCNT_ENUM (mezcla, anti-aliasing, test de transparencia, etc).
- void glFlush (u32 mode): espera al barrido vertical e intercambia los buffers de geometría y de renderizado. Se puede especificar el modo de almacenamiento de los vértices (por Z o por W) y el modo de ordenación de los polígonos translúcidos (automático o manual).

Por otra parte, se dispone de una función con la que definir la ventana de visualización.

- void glViewport (uint8 x1, uint8 y1, uint8 x2, uint8 y2): especifica el tamaño y posición de la pantalla donde se va a realizar el pintado. Se puede configurar varias veces por ciclo. A continuación se especifican las funciones que definen el frustrum. Existen dos versiones de la misma función, una de punto flotante y otra d f32. Como en anteriores casos, se recomienda emplear la versión f32.
- void glFrustum (float left, float right, float bottom, float top, float near, float far): especifica el volumen de visualización o frustrum para la matriz de proyección. Versión de punto flotante.
- void glFrustumf32 (int left, int right, int bottom, int top, int near, int far): especifica el volumen de visualización o frustrum para la matriz de proyección. Versión f32.

También se ponen a disposición funciones para configurar el fondo, tanto su color como su profundidad o su identificador.

- void glClearColor (uint8 red, uint8 green, uint8 blue, uint8 alpha): establece el color del fondo.
- void glClearDepth (fixed12d3 depth): establece la profundidad del fondo. Normalmente se establece en el valor máximo, es decir GL_MAX_DEPTH o, lo que es equivalente, 7FFFh.
- void glClearPolyID (uint8 ID): establece el identificador de polígono del fondo. Debe tener un valor reservado para que los estados que dependen de este parámetro (blending, transparencia, anti-aliasing, etc) funcionen correctamente.

Además, se dispone de una función para determinar una distancia a partir de la cual dejar de pintar polígonos.

- void glCutoffDepth (fixed12d3 wVal): establece una distancia a la cámara a partir de la cual se dejan de pintar polígonos.

La siguiente función define el valor de referencia de transparencia por debajo del cual no se pintan los polígonos.

- void glAlphaFunc (int alphaThreshold): establece el valor de transparencia de referencia. Los polígonos cuya transparencia esté por debajo de este valor no se pintarán.

Hay también a disposición funciones para configurar la cámara virtual, de forma que se definen su posición en el espacio y el punto al que apunta. Como en otras ocasiones, ante las versiones de punto flotante y f32, es preferible usar la última.

- void gluLookAt (float eyex, float eyey, float eyez, float lookAtx, float lookAty, float lookAtz, float upx, float upy, float upz): establece la posición (eyex, eyey, eyez), el lugar al que apunta (lookAtx, lookAty, lookAtz) y un vector unitario que define qué dirección es la cenital para la cámara (upx, upy, upz). Versión de punto flotante.

- void gluLookAtf32 (int eyex, int eyey, int eyez, int lookAtx, int lookAty, int lookAtz, int upx, int upy, int upz): establece la posición (eyex, eyey, eyez), el lugar al que apunta (lookAtx, lookAty, lookAtz) y un vector unitario que define qué dirección es la cenital para la cámara (upx, upy, upz). Versión f32.

Para configurar la iluminación, y más concretamente, cada una de las cuatro luces que se pueden emplazar en la escena, se dispone de una función.

- void glLight (int id, rgb color, v10 x, v10 y, v10 z): establece los parámetros de la luz determinada por su identificador (id). Éstos son su color y su vector de dirección. Este último parámetro indica que las luces sólo pueden ser direccionales, es decir de rayos paralelos.

Como ya se ha comentado, un efecto disponible en la Nintendo DS es la niebla. Ésta sirve para difuminar los objetos lejanos.

La niebla se calcula mediante una tabla de 32 valores de densidad. La profundidad de la niebla la determina FOG_OFFSET, y su variación FOG_SHIFT. Con estos parámetros se determina la profundidad de cada valor de la tabla. De esta forma, la densidad de la niebla para un píxel se determina contrastando su profundidad con esta tabla: se establece entre qué posiciones de la tabla se encuentra y se realiza una interpolación lineal de los valores de densidad de éstas.

Las siguientes funciones permiten configurar dicho efecto.

- void glFogColor (uint8 red, uint8 green, uint8 blue, uint8 alpha): establece el color de la niebla y su transparencia.
- void glFogDensity (int index, int density): establece la densidad para una posición de la tabla determinada.
- void glFogOffset (int offset): establece la profundidad de la niebla.
- void glFogShift (int shift): establece la variación de la niebla.

Una de las características gráficas de la Nintendo DS es su modo sombreado toon. Éste consiste en que la variación del sombreado no es uniforme, sino brusca. Se podría decir que el sombreado estándar, que en este caso es el llamado Gouraud, hay una variación continua (aunque esto evidentemente no es así, ya que se trata de un sistema digital donde hay una precisión limitada) y el sombreado toon es discreto, con varios rangos de variación intentando imitar el sobreado de los dibujos animados.

De esta forma, la iluminación sobre un modelo en este modo se calcula mediante una tabla con dichos rangos. Cada rango define, por un lado, dos niveles de rojo que lo limitan (el nivel de rojo es el que es el que se emplea para calcular este sombreado), y por otro lado, el color de que ha de asignarse en dicho rango. Por lo tanto, para calcular el color en un pixel se extrae su nivel de rojo, se contrasta con la tabla, determinando el rango en el que se encuentra, obteniendo así el color que hay que aplicar. Dado que el nivel de rojo varía entre el rango [0, 31], el rango máximo de esta tabla varía también en este intervalo.

Las siguientes funciones permiten configurar este sombreado.

- void glSetToonTable (const uint16 *table): establece una tabla de sombreado toon externa pre-configurada.
- void glSetToonTableRange (int start, int end, rgb color): establece un rango de la tabla de sombreado toon y su color correspondiente.

Por último, en las funciones de configuración se encuentra una función para definir el color del contorno para un identificador determinado. Se pueden definir hasta 8 colores de contorno, abarcando cada uno de ellos 8 identificadores de polígono. De esta forma, el color de contorno de los polígonos con identificador entre 0 y 7 será el primero, el de los polígonos con identificador entre 8 y 15 será el segundo y así sucesivamente.

- void glSetOutlineColor (int id, rgb color): establece el color del contorno asociado a su identificador.

Utilidades

Hay en videoGL una serie de funciones que aportan utilidades de diversa índole. Dos de ellas sirven para obtener los valores de ciertos parámetros, según su tipo.

- void glGetFixed (const GL_GET_ENUM param, int *f): devuelve un valor de punto fijo correspondiente al valor de diversos estados del hardware gráfico, cuyo parámetro viene definido por el elemento de GL_GET_ENUM introducido.
- void glGetInt (GL_GET_ENUM param, int *i): devuelve un valor entero correspondiente al valor de diversos estados del hardware gráfico, cuyo parámetro viene definido por el elemento de GL_GET_ENUM introducido.

La última de estas funciones de utilidad se encarga de enviar las denominadas display lists. Éstas consisten en paquetes de comandos que se envían de golpe para reducir el tiempo de transferencia y que puedan entregarse al hardware gráfico de forma paralela a otros procesados que esté haciendo la CPU.

- void glCallList (const u32 *list): envía una display lists (lista de comandos empaquetados) directamente al FIFO de gráficos.

4.3.2. Enumeradores de parámetros

En videoGL se ponen a disposición una serie de enumeradores (como tipo de dato), que se emplean tanto para definir los parámetros de ciertos registros del motor 3D, como para definir modos de funcionamiento. De esta forma, pasándole estos elementos a las funciones que requieren el tipo e dato correspondiente, se accede a los parámetros del motor.

Ciertos enumeradores son simplemente tipos definidos que agrupan cadenas de caracteres constantes para, internamente, diferenciar entre un modo u otro y actuar en consecuencia. Otros, sin embargo, asignan a estas constantes un valor que hace referencia al bit del parámetro contenido en un registro. Así, éstos se emplean para acceder directamente a dicho parámetro, a través de la función que maneja el registro correspondiente o a través de la dirección del registro directamente (en ciertas ocasiones es necesario bajar a más bajo nivel).

DISP3DCNT_ENUM

- GL_TEXTURE_2D = 1<<0
- GL_TOON_HIGHLIGHT = 1<<1

- GL_ALPHA_TEST = 1<<2
- GL_BLEND = 1<<3
- GL_ANTIALIAS = 1<<4
- GL_OUTLINE = 1<<5
- GL_FOG_ONLY_ALPHA = 1<<6
- GL_FOG = 1<<7
- GL_COLOR_UNDERFLOW = 1<<12
- GL_POLY_OVERFLOW = 1<<13
- GL_CLEAR_BMP = 1<<14

Contiene accesos a los parámetros del registro de control de visualización 3D. Cada uno de los parámetros está definido en el registro por un bit, al cual se accede a través del valor asignado a cada elemento de la enumeración. Así, GL_TEXTURE_2D opera sobre el primer bit del registro, GL_TOON_HIGHLIGHT sobre el segundo (1<<1 equivale a 10 en binario, apuntando así al segundo bit) y así sucesivamente. De esta forma, con valores múltiplos de 2 (en este caso se consiguen mediante desplazamientos de bit a la izquierda) se puede operar directamente sobre cada uno de los bits que representan a los parámetros de este registro. Para activar o desactivar cada una de las opciones de visualización se llama a las funciones *glEnable()* y *glDisable()* respectivamente, pasándole la propiedad que se desea habilitar.

GL_GET_ENUM

- GL_GET_VERTEX_RAM_COUNT
- GL_GET_POLYGON_RAM_COUNT
- GL_GET_MATRIX_VECTOR
- GL_GET_MATRIX_POSITION
- GL_GET_MATRIX_PROJECTION
- GL_GET_MATRIX_CLIP
- GL_GET_TEXTURE_WIDTH
- GL_GET_TEXTURE_HEIGHT

Contiene constantes con las que definir qué clase de información se quiere obtener de la llamada a las funciones *glGetInt()*, *glGetFixed()*.

GL_GLBEGIN_ENUM

- GL_TRIANGLES = 0
- GL_QUADS = 1
- GL_TRIANGLE_STRIP = 2
- GL_QUAD_STRIP = 3
- GL_TRIANGLE = 0

- GL_QUAD = 1

Contiene constantes con las que definir cómo realizar el pintado de polígonos en el momento del inicio de la lista de vértices, mediante la llamada a la función *glBegin()*.

GL_MATERIALS_ENUM

- GL_AMBIENT = 0x01
- GL_DIFFUSE = 0x02
- GL_AMBIENT_AND_DIFFUSE = 0x03
- GL_SPECULAR = 0x04
- GL_SHININESS = 0x08
- GL_EMISSION = 0x10

Contiene constantes con las que definir qué propiedad material se quiere asignar mediante la función *glMaterialf()*.

GL_MATRIX_MODE_ENUM

- GL_PROJECTION = 0
- GL_POSITION = 1
- GL_MODELVIEW = 2
- GL_TEXTURE = 3

Contiene los modos de matriz disponibles, activados mediante la función *glMatrixMode()*. De ésta forma se especifica con qué matriz se va a operar a continuación.

GL_POLY_FORMAT_ENUM

- POLY_FORMAT_LIGHT0 = 1<<0
- POLY_FORMAT_LIGHT1 = 1<<1
- POLY_FORMAT_LIGHT2 = 1<<2
- POLY_FORMAT_LIGHT3 = 1<<3
- POLY_MODULATION = 0<<4
- POLY_DECAL = 1<<4
- POLY_TOON_HIGHLIGHT = 2<<4
- POLY_SHADOW = 3<<4
- POLY_CULL_FRONT = 1<<6
- POLY_CULL_BACK = 2<<6
- POLY_CULL_NONE = 3<<6
- POLY_FOG = 1<<15

Contiene accesos a los parámetros del registro de formato de polígono del motor, bit a bit,

para su activación o desactivación. Éstos se asignan llamando a la función *glPolyFmt()*, pasándole la suma binaria de los parámetros que se quieran aplicar.

GL_TEXTURE_PARAM_ENUM

- GL_TEXTURE_WRAP_S = 1<<16
- GL_TEXTURE_WRAP_T = 1<<17
- GL_TEXTURE_FLIP_S = 1<<18
- GL_TEXTURE_FLIP_T = 1<<19
- GL_TEXTURE_COLOR0_TRANSPARENT = 1<<29
- TEXGEN_OFF = 0<<30
- TEXGEN_TEXCOORD = 1<<30
- TEXGEN_NORMAL = 2<<30
- TEXGEN_POSITION = 3<<30

Contiene accesos a los elementos del registro de parámetros de textura, bit a bit. Éstos se asignan llamando a la funciones *glTexImage2d()*, en la creación de la textura, o *glTexParameter()*, en la definición de parámetros adicionales, pasándole la suma binaria de los parámetros que se quieran aplicar.

GL_TEXTURE_SIZE_ENUM

- TEXTURE_SIZE_8 = 0
- TEXTURE_SIZE_16 = 1
- TEXTURE_SIZE_32 = 2
- TEXTURE_SIZE_64 = 3
- TEXTURE_SIZE_128 = 4
- TEXTURE_SIZE_256 = 5
- TEXTURE_SIZE_512 = 6
- TEXTURE_SIZE_1024 = 7

Contiene los diferentes tamaños en píxels que puede tener una textura, para pasárselo a las funciones *glTexImage2d()* o *glTexParameter()* y así asignar éste parámetro a la textura correspondiente.

GL_TEXTURE_TYPE_ENUM

- GL_RGB32_A3 = 1
- GL_RGB4 = 2
- GL_RGB16 = 3
- GL_RGB256 = 4
- GL_COMPRESSED = 5

- GL_RGB8_A5 = 6
- GL_RGBA = 7
- GL_RGB = 8

Contiene los tipos de textura con los que definir este parámetro de la textura correspondiente mediante las funciones *glTexImage2d()* o *glTexParameter()*.

GLFLUSH_ENUM

- GL_TRANS_MANUALSORT = (1 <<0)
- GL_WBUFFERING = (1 <<1)

Contiene accesos a los parámetros del registro de intercambio de los buffers de geometría y de render. Éstos se aplican con la llamada a la función *glFlush()*, pasándole la suma binaria de los parámetros que se quieran aplicar.

4.4. Procedimiento general en el uso del motor 3D

Vistos los diferentes módulos que componen la librería Libnds, que es la que permite acceder a las distintas funcionalidades de la consola, en este epígrafe se pretende plasmar una visión general de la utilización de ésta en una aplicación 3D en la Nintendo DS, siempre teniendo en cuenta que el objetivo es el pintado de objetos tridimensionales, y no se contemplan el resto de funcionalidades como el sonido, el Wifi, etc.

En el diagrama de la Figura 4.1 se ilustra el proceso general y más básico que ha de implementarse para realizar una aplicación 3D. Dado que es una visión generalista, este proceso es muy discutible y personalizable, dependiendo de las características de la aplicación.

La primera etapa del proceso es la inicialización, tanto del hardware de la consola, como del módulo videoGL, para su utilización. Ésta consiste en definir en qué pantalla visualizar el contenido del motor principal, definir el funcionamiento de dicho motor (en este caso será en modo 3D) y por último iniciar el núcleo GL para poner en marcha los parámetros correspondientes y poder llamar a sus funciones.

El siguiente paso sería la configuración del motor 3D, y más en particular, del registro de control de visualización 3D. De esta forma, se activan los procesos o estados que se quieran emplear (blending, anti-aliasing, sombreado toon, etc). Además, se define la posición y el tamaño de la ventana de visualización, se configura la matriz de proyección, y se posiciona la cámara virtual. Además, se puede establecer el parámetro de transparencia de referencia, las luces que se van a colocar en la escena y los parámetros del fondo. Por último, en esta etapa se pueden definir los parámetros del sombreado toon, del contorno y la tabla de la amplitud del resalte especular.

Como ya se ha explicado, este proceso puede variar dependiendo de la aplicación, pudiéndose implementar antes del bucle principal o durante éste.

Tras esta configuración se asignarían los bancos de la VRAM que se van a emplear para texturas, sprites o paletas.

A continuación, y siempre teniendo en cuenta que es un proceso modificable, se generaría

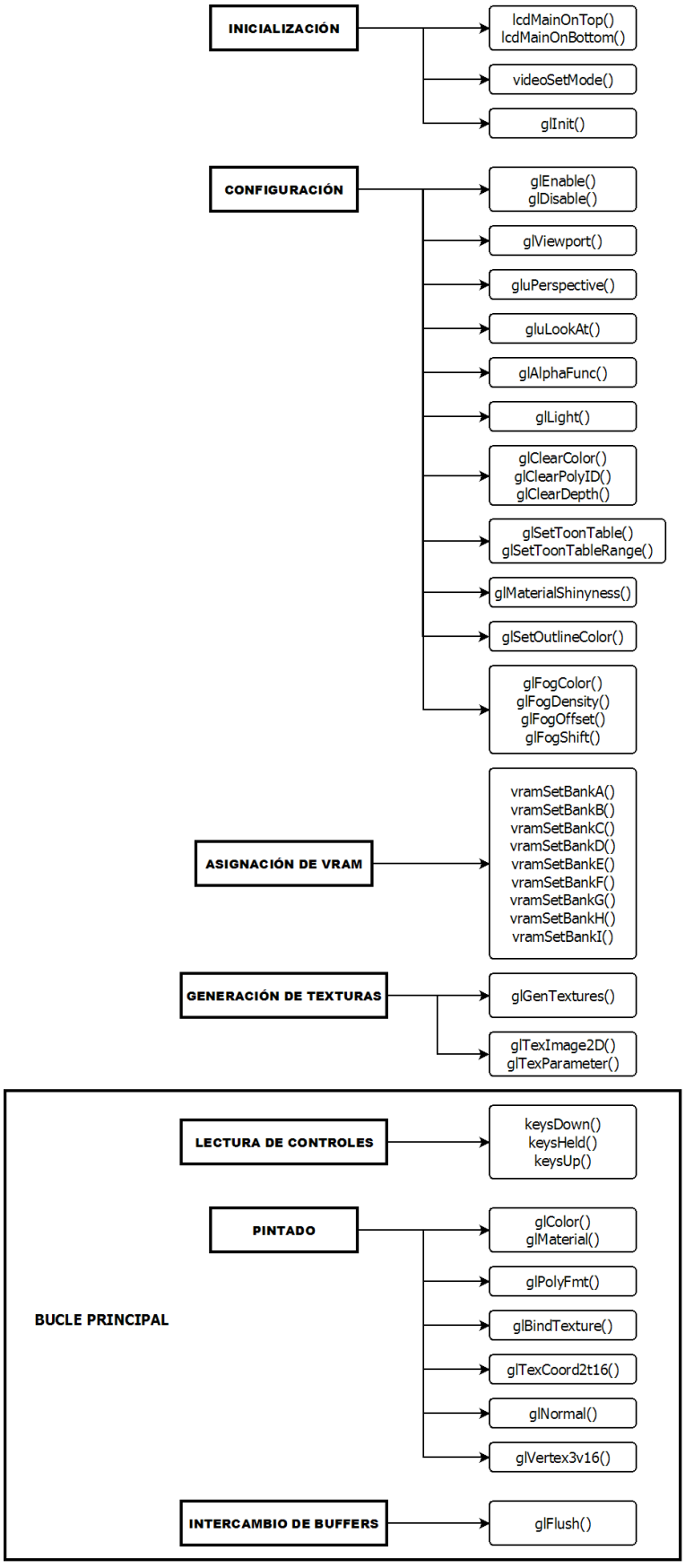

Figura 4.1: Diagrama del proceso general de implementación de una aplicación 3D.

la lista de texturas y las texturas en sí, a partir de las imágenes cargadas en memoria, estableciendo sus atributos.

El proceso entraría a continuación en el bucle principal, donde se controlan las acciones externas del usuario y se realiza el pintado. Éste último, como ya se ha explicado, consiste en, por un lado, definir los colores del polígono (ya sea por color directo o por su material), sus atributos y su textura correspondiente y por otro, para cada vértice del polígono, mandar sus coordenadas de textura, sus normales y sus vértices.

El proceso finalizaría intercambiando los buffers de geometría y de render, con el fin de actualizar la información visual a cada ciclo.

Capítulo 5

Trabajo previo al desarrollo

En general, un proyecto requiere de una fase de investigación previa en la que se contrastan los objetivos marcados con los recursos disponibles, para así poder trazar las vías de actuación que se van a seguir en el desarrollo de éste. En este proyecto concretamente, esta fase ha sido esencial para alcanzar los objetivos de la manera más eficiente posible y evitando dar rodeos innecesarios a mitad de desarrollo.

En este capítulo se describirá, por lo tanto, cómo ha sido esta fase inicial, los pasos que se han seguido, las barreras que se han encontrado y las conclusiones y decisiones que se han sacado. Todo ello dejando bien claro el proceso y el entorno en el que se ha trabajado en cada momento.

Además, a partir de esta fase previa, se han definido una serie de herramientas necesarias para el manejo de datos en formatos específicos, una de ellas, el conversor a EDL, propia de este proyecto, y otras, de conversión de formatos de imagen, externas a éste.

En la Figura 5.1 y la Figura 5.2 se muestran esquemas de los diferentes entornos del proyecto en una primera aproximación a éste y en el escenario final respectivamente, distinguiendo según una gama de colores los programas externos al proyecto, los realizados en el entorno PC, y los creados específicamente en el desarrollo de este proyecto (el objetivo principal), especificando el flujo de datos y los formatos de éstos en el paso de un programa a otro.

5.1. Primera aproximación

En el desarrollo de una herramienta de renderizado de gráficos, no basta con definir un entorno de desarrollo de programación, como se he hecho en el capítulo anterior. Es necesario, además, disponer de los recursos gráficos para poder realizar las pruebas pertinentes de cada funcionalidad implementada. Por ello, se ha tenido que realizar la elección del programa de modelado, teniendo en cuenta las necesidades y los objetivos del proyecto.

Por otro lado, una vez hecha este elección, se han tenido que definir una vía de actuación que permita adaptarse a estos recursos y a partir de la cual empezar a diseñar la aplicación de render en la Nintendo DS .

Por lo tanto, en este apartado se describirá el proceso de desarrollo previo (y en algunos

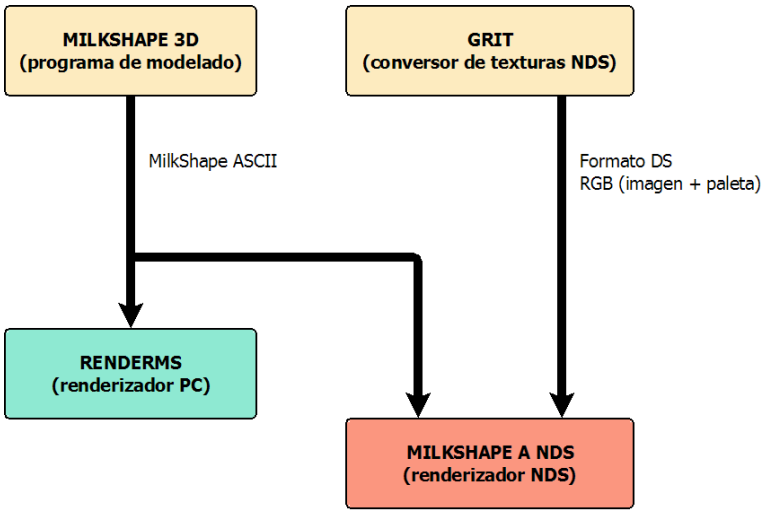

Figura 5.1: Esquema del entorno en una primera aproximación. En color amarillo aparecen los programas externos al proyecto empleados en éste; en azul la herramienta de renderizado implementada en el entorno PC; en rojo aparece la herramienta de renderizado del entorno de la Nintendo DS, desechada en esta fase inicial por motivos de eficiencia.

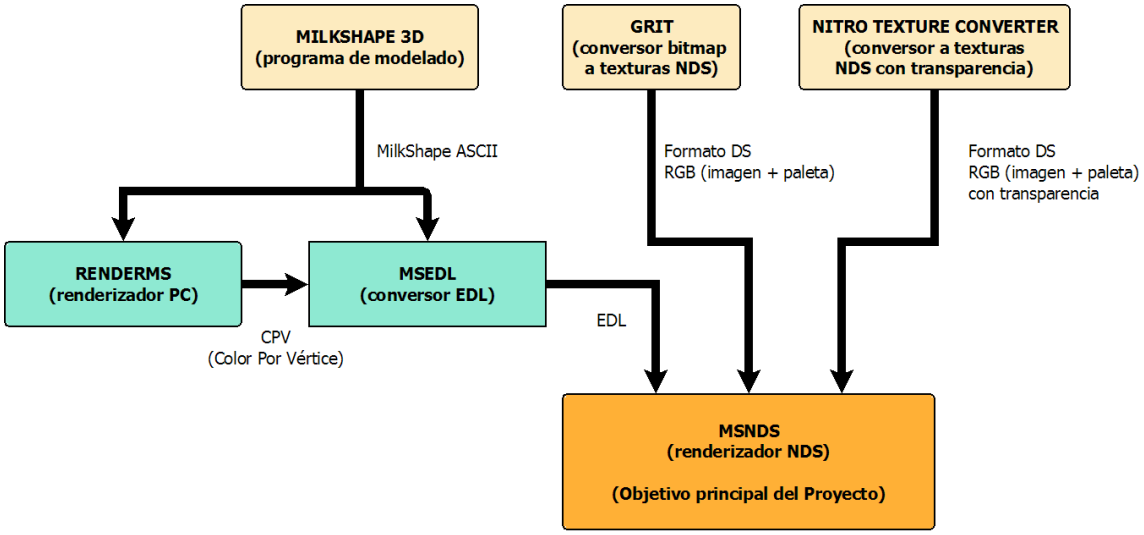

Figura 5.2: Esquema del entorno final empleado a lo largo del proyecto. En color amarillo aparecen los programas externos al proyecto empleados en éste; en azul la herramienta de renderizado y el conversor al formato EDL implementados en el entorno PC; en naranja aparece la herramienta final de renderizado del entorno de la Nintendo DS, que supone el objetivo primordial del proyecto, centrándose el desarrollo en la implementación de ésta.

casos paralelo) al desarrollo del motor de render.

5.1.1. Elección de la herramienta de modelado

Con el fin de realizar las pruebas y posteriores demostraciones en el motor de render implementado, es necesario disponer de una serie de modelos que incorporen las características gráficas adecuadas para cada una de las funcionalidades del renderizador. Por lo tanto, para su creación, es necesario emplear un programa de modelado que ofrezca

dichas características.

Si bien, cómo ya se ha comentado en el capítulo referente a los gráficos por ordenador, existen multitud de herramientas de modelado, algunas de ellas realmente potentes, para este proyecto, en el que el objetivo no es la calidad de los modelos en sí mismos, si no las capacidades de la Nintendo DS para su visualización, se ha seleccionado una muy sencilla y accesible para los recién iniciados en este campo: MilkShape 3D.

MilkShape 3D, de chUmbaLumsOft [14], es un software de modelado orientado a personajes animados con esqueleto con bajo número de polígonos, fácil de usar, con unas opciones limitadas que favorecen el entendimiento de la herramienta y su accesibilidad. Por ello, el interfaz es muy sencillo, con unas pocas funcionalidades a la vista, evitando, de esta forma, el rechazo del usuario no experimentado. Dada su sencillez y que, a pesar de sus limitadas opciones, aporta las características requeridas en este proyecto, es la herramienta idónea para la creación de los modelos destinados a ser pintados en la Nintendo DS.

Con MilkShape 3D se puede dotar a los modelos con las propiedades necesarias para comprobar las capacidades de la Nintendo DS. La creación de las mallas se puede realizar mediante modelado poligonal, es decir, operando directamente sobre los vértices y las caras, para obtener la forma deseada. Una forma sencilla de obtener formas complejas es partir de un volumen prediseñado, como cubos, esferas o planos, e ir variando su forma mediante traslaciones, escalados, rotaciones y extrusiones. Las mallas en sí se crean organizando los polígonos por grupos, con un nombre específico para cada uno, de forma que se pueden asignar atributos diferentes, como materiales, a grupos de vértices distintos.

En lo que se refiere a los materiales, éstos se pueden modelar mediante las componentes básicas: color ambiente, color difuso, color especular, color emisivo, transparencia y amplitud del resalte especular. Esto es lo idóneo, ya que estos son los parámetros que se pueden especificar y controlar en la Nintendo DS (excepto el último, que se controla mediante una tabla predefinida).Además, un aspecto interesante de los materiales en MilkShape 3D es la posibilidad de asociar al material dos texturas, capacidad que aprovechará el motor de render. El material se especifica por su nombre y va asociado a una malla o grupo, no soportando color por vértice.

Por último, la animación es por esqueleto, creando articulaciones, definidas por nombre, y efectuando la jerarquización entre ellas. Posteriormente, es posible realizar el pesado de los vértices asociados a cada articulación. La animación se realiza especificando el número de cuadros por animación y definiendo poses en claves de animación seleccionadas. Las articulaciones solo se pueden rotar y trasladar, afectando la transformación únicamente a las articulaciones más distales en la jerarquía (las más externas). Así, la transformación final de una articulación afecta a los vértices que tiene asociados, generando movimiento en las mallas. Por lo tanto, es un sistema básico de animación por esqueleto.

Una funcionalidad importante del MilkShape 3D, que más tarde se empleará de forma sistemática, es la de asignar comentarios a los modelos completos, a las mallas, a los materiales, y a los huesos. De esta forma, cuando se extraiga la información del modelo, que se verá más adelante, se pueden asociar ciertas opciones a dichos atributos.

Una vez determinada la herramienta de modelado que se va a emplear, ya se pueden crear los modelos que se van a cargar en la Nintendo DS. Para esto, es necesario exportar el modelo a un tipo de fichero que pueda ser leído, desde el cual extraer toda esta información. La opción más básica es exportar al formato MilkShape ASCII.

5.1.2. Formato MilkShape ASCII

Con el objetivo de cargar los modelos creados con MilkShape 3D en la Nintendo DS se ha determinado emplear el formato MilkShape ASCII como primer soporte para manejar la información de los modelos 3D.

Este formato consiste en un fichero de texto. Éste contiene, de forma ordenada, toda la información referente al modelo. Primero se especifica el número de frames y de mallas. Para cada malla, se especifica:

- su nombre
- el número de vértices
- la lista de vértices (cada uno de ellos compuesto por su posición, sus coordenadas de textura y el índice del hueso que lo controla)
- el número de normales
- la lista de normales
- el número de triángulos
- la lista de triángulos (cada uno de ellos compuesto por los índices de los vértices que los componen).

A continuación, se especifica el número de materiales. Para cada material se especifica:

- sus componentes del material (color ambiente, color difuso, color especular, color emisivo)
- su amplitud del resalte especular
- su transparencia
- el nombre del archivo de imagen de la primera textura
- el nombre del archivo de imagen de la segunda textura

Seguidamente, se especifica el número de articulaciones, seguido de las propias. En cada articulación se detalla:

- su nombre
- el nombre de su "padre"
- su posición
- el número de claves de posición
- la lista de claves de posición
- el número de claves de rotación
- la lista de claves de rotación

Finalmente, se especifican el número de comentarios de grupo, de material, de articulación y de modelo, seguidos de éstos.

En el proyecto, se ha tenido a disposición una herramienta de render que ha sido empleada a modo de referencia, y en paralelo al desarrollo: **renderms**. Esta herramienta, implementada por el director de este proyecto, Enrique Rendón, pertenece al dominio del PC, siendo un renderizador basado en OpenGL. Este programa recibe un archivo de texto MilkShape ASCII y, en tiempo de ejecución, lo lee (siguiendo las pautas men-

cionadas anteriormente) y almacena en las estructuras correspondientes la información relativa al modelo y sus atributos. Una vez almacenados estos datos, la aplicación accede a ellos para realizar el pintado del modelo con una serie de opciones configurables. De esta forma, a partir de un fichero en formato MilkShape ASCII se realiza el render de un modelo creado con MilkShape 3D.

Siguiendo como referencia esta aplicación, y ya dentro del marco de este proyecto, se implementó un lector de MilkShape ASCII junto con un renderizador básico, esta vez enfocado a la Nintendo DS. No se entrará en detalle en su implementación en este texto dado que, por motivos que se comentarán a continuación, este sistema de carga de modelos fue descartado.

El proceso es muy similar al realizado en renderms: se carga el fichero del modelo guardado en la memoria principal de la consola y se prepara para su lectura. Ésta consiste en ir siguiendo el protocolo de ordenación de los datos y en cada línea leída almacenar la información correspondiente. Estos datos se almacenan en una estructura de modelo, que más tarde se pasa al renderizador que pinta el modelo en pantalla. Este renderizador era un simple prototipo para probar la funcionalidad del sistema, siendo sus capacidades muy limitadas (pintado de polígonos con materiales y texturas).

Si bien el sistema funcionaba correctamente, ya que se extraían adecuadamente los datos del modelo y se pintaban perfectamente, éste era muy ineficiente. La Nintendo DS no tiene un procesador lo suficientemente rápido como para leer y procesar tal cantidad de datos de forma "instantánea". Hay que tener en cuenta que la lectura implica leer y almacenar cada vértice, cada normal y cada triángulo de cada una de las mallas del modelo, además de su información de materiales (los datos de animación no estaban implementados en este punto del desarrollo). Si el número de éstos es considerable, y no se requieren valores muy altos para esto, la cantidad de procesos de lectura/escritura se eleva mucho, siendo además operaciones costosas. Por lo tanto, cada vez que se efectúa la carga de un modelo para su pintado, ésta puede tardar entre varios segundos a un minuto, dependiendo del tamaño del archivo. Además, el almacenamiento del modelo se efectúa de forma poco eficiente en términos de memoria (aunque también afecta a la problemática anterior): los vértices, normales, y triángulos se almacenan en vectores, cuya longitud depende del número de dichos parámetros (siendo, por otra parte, inevitable, ya que se trata de un fichero de texto plano y es la única forma de acceder a los datos). Esto, evidentemente, es un consumo excesivo de la preciada memoria de la consola.

Por estas razones de ineficiencia, se decidió implementar un formato propio, cuyos datos estén preparados para la Nintendo DS y, además, tener la posibilidad de personalizar la información incluida en éste para mejorar su cohesión con el motor de render desarrollado.

5.1.3. Entorno PC

Llegados a este punto, es necesario comentar y diferenciar claramente las diferentes vías de desarrollo que se han seguido en este proyecto. El objetivo de este proyecto es estudiar y, por medio de la implementación de una aplicación de render, aprovechar las capacidades gráficas de la Nintendo DS. Por lo tanto, el grueso del trabajo efectuado ha ido exclusivamente en esta línea. Sin embargo, para realizar una aplicación eficiente, ha sido necesario, como ya se ha comentado, realizar en paralelo el desarrollo de una herramienta propia con la que generar ficheros en un formato específico para la Nintendo DS.

Esto quiere decir que, dentro del marco del proyecto, entra la investigación referente a la Nintendo DS y el desarrollo de la herramienta de render. Por otro lado, y en una vía de desarrollo aparte, se ha implementado el conversor al formato específico de la consola. Esta vía paralela al desarrollo del proyecto en sí, imprescindible para su ejecución, se denominará el entorno PC, quedando así el marco de este proyecto como entorno DS.

Por lo tanto, mientras en el entorno PC se ha ido creando la herramienta de conversión al formato adaptado a la Nintendo DS, en el entorno DS se ha ido adaptando el renderizador a dicho formato.

5.2. Extended Display List: un formato adaptado a la Nintendo DS

El **Extended Display List (EDL)** es un formato de archivo que alberga la información de modelos 3D creados con MilkShape 3D, completamente adaptado a la Nintendo DS, almacenando muchos de los datos en el formato esperado por el hardware gráfico ésta. Éste ha sido concebido y desarrollado por Enrique Rendón específicamente para este proyecto, en función de los objetivos y las necesidades de éste.

De esta forma, se ha tenido en cuenta el marco de desarrollo a la hora de diseñar el formato, con el objetivo de aunar en un solo archivo, no solo la información del modelo como sus vértices, normales o materiales, sino ciertas opciones de visualización, con el fin de mejorar la renderización, aplicar efectos o mejorar la eficiencia del pintado.

Por lo tanto, en este epígrafe se justificará su creación y las ventajas que aporta al proyecto, para pasar a especificar en detalle el formato en sí.

5.2.1. Justificación y descripción general

En la escena *homebrew* existen diversos formatos con los que cargar el material 3D en la Nintendo DS. El más común es el que almacena una *display list* con los comandos precisos para renderizar el modelo en cuestión. Este archivo se genera mediante un *plugin*, Blender NDS Exporter [5] que, añadido a Blender (el otro software de modelado open source por excelencia), importa los datos del modelo con dicha estructura. De esta forma, en la aplicación 3D, tan solo hay que llamar a la función *glCallList*() pasándole la dirección del modelo cargado en memoria.

Este sistema es muy eficiente y sencillo de implementar. Sin embargo, el pintado del modelo viene definido de antemano por el formato, por lo que deja poco margen para diseñar un sistema de render propio. Además, este método no soporta animación, dado que el envío de comandos viene prefijado sin tener en cuenta transformaciones. Por lo tanto, no es un formato idóneo con el que manejar los modelos en este proyecto, dado que se va a diseñar completamente el sistema de render y se requiere un gran nivel de control.

EDL surge pues como alternativa a este formato, compartiendo la característica de que almacena datos preparados para la Nintendo DS, en los formatos con los que ésta trabaja. Sin embargo no es una *display list* como tal, puesto que no contiene comandos empaquetados, sino información propia del modelo.

Por otra parte, EDL contiene información referente a opciones de render. Esto significa que, en la fase de modelado, mediante los comentarios añadidos a los diferentes conjuntos del modelo, se pueden especificar ciertos parámetros del pintado del modelo en tiempo de render. Esta es una funcionalidad muy potente, ya que el propio modelo contiene toda la información correspondiente a la manera en que tiene que ser procesado. Sin embargo estas opciones siempre pueden ser activadas o desactivadas por el usuario de la herramienta de render, como se explicará en el capítulo referente al desarrollo de ésta.

La generación de los archivos en este formato se realiza mediante la herramienta de conversión específica de éste: **ms2edl**. Su función es generar un archivo binario en formato EDL a partir de los datos contenidos en un archivo MilkShape ASCII. A ésta se le especifica el archivo de MilkShape ASCII que se desea convertir y la aplicación almacena tanto los datos del modelo como las opciones de pintado, según la estructura del formato. Además del archivo, a la aplicación hay que especificarle la versión de EDL a la cual se quiere exportar el modelo, dado que existen varias, las cuales van incorporando más funcionalidades y opciones.

Como conclusión, EDL es un formato propio, diseñado según las necesidades y los objetivos que define el marco de desarrollo del proyecto, lo que, combinado con el hecho de que se ha implementado la herramienta de render teniendo en cuenta las especificaciones de este formato, lo convierten en una plataforma idónea para almacenar modelos 3D en este proyecto.

5.2.2. Especificaciones del formato EDL

El formato EDL es un formato binario, y como tal, tiene los datos estructurados de una cierta forma. Es esencial conocer esta estructura a la hora de implementar un lector que siga un orden concreto y almacene correctamente cada elemento en el contexto específico. Por ello es necesario conocer en detalle la especificación del formato EDL, dado que la herramienta de render se basa en éste.

Los datos en EDL están agrupados en palabras de 32 bits. De esta forma se evitan problemas de alineamiento, ya que la lectura se realiza siempre tomando este número de bits. Así, tan solo hay que saber qué contienen cada uno de estos bloques de 32 bits y procesarlos en consecuencia. Este sistema facilita la lectura pero, dado el diseño de la estructura de datos, en ocasiones se desaprovecha espacio enviando bytes sin información alguna, aunque esto se produce la menor de las veces y en situaciones muy puntuales. A partir de este punto, cuando se hable de una palabra o un parámetro, se referirá a una palabra de 32 bits.

En este texto, no se entrará en detalle en cuanto a cómo están organizados los datos en estas palabras de 32 bits, si bien esta información está disponible en el Anexo A. El objeto de este apartado es tan solo especificar las opciones de las que dispone el formato EDL y cómo están organizados los datos de forma global.

Los datos en EDL se agrupan en sectores que representan diferentes propiedades del modelo 3D. De esta forma se van leyendo los datos referentes a cada componente del modelo de forma sucesiva y ordenada. El orden en que dichos sectores vienen almacenados es: datos de modelo, datos de esqueletos, datos de articulaciones, datos de mallas y por último datos de materiales. Adicionalmente, a modo de cabecera, se incluye una palabra especificando, por un lado, que se trata de un archivo EDL, y por otro, el número de

versión de éste, dado que existen cuatro versiones diferentes, a cada cual más completa en información y opciones. La información que se detalla a continuación corresponde a la última versión de EDL.

Modelo

El formato EDL la primera información que aporta es referente el modelo general. De esta forma, se proporcionan los datos correspondientes a los componentes del modelo:

- Su **número de materiales**
- Su **número de mallas**
- Su **número de esqueletos**
- Su **número de articulaciones**

Éstos se emplearán en la fase de render para saber cuántas mallas hay que pintar o el número de elementos que hay que comprobar en la búsqueda de materiales.

Además, almacena las opciones de render que se hayan establecido a través de los comentarios en MilkShape 3D:

- La presencia de **materiales reflectivos** (necesario para realizar o no una capa de pintado de reflexión esférica)
- La presencia de algún material con **segunda textura** (necesario para realizar o no una capa de segunda textura)
- **Orden de pintado** de las capas de **reflexión** y **segunda textura**

Estas opciones se almacenan en forma de banderas o *flags*, es decir, con un bit que indica un estado u otro.

Esqueletos

En el capítulo de introducción a los gráficos por ordenador se ha explicado que la animación de un modelo se efectúa asociando a éste un esqueleto, compuesto por diferentes huesos y articulaciones, que son los que realizan el movimiento, imprimiendo una deformación a la malla.

En cuanto al sistema de animación, conviene explicar algunos puntos. Cómo ya se ha comentado, una animación consiste en una serie de claves de animación que especifican ciertos atributos (en el caso más sencillo la posición y la rotación) en un instante de tiempo concreto. En este caso, el tiempo se mide en frames, dado que se suele especificar la tasa de frames por segundo, de esta forma si se considera el frame 30 a 60 fps esto equivale a 0,5 segundos. De esta forma, generando dichas claves y realizando una interpolación lineal de las componentes del movimiento entre claves se define el movimiento a lo largo del tiempo.

Un punto a tener en cuenta es que en un solo clip de animación pueden estar contenidas varias costumbres de animación que definen diferentes tipos de movimiento, como pueden ser andar, correr o saltar, para dar variedad a la animación de los objetos. Esto se hace así ya que a un modelo o esqueleto tan solo se le puede asignar un único clip de animación. De esta forma, una costumbre de animación se define por sus frames inicial y final, creando

de esta forma sub-animaciones.

MilkShape 3D no exporta la información de movimiento en cada frame de la animación, sino únicamente la posición y la rotación de la articulación en las claves de animación creadas. Sin embargo, para facilitar la tarea de animación en el renderizador, el conversor EDL calcula la posición y la rotación en cada frame de la animación, atendiendo al número de frames de ésta y una velocidad de muestreo establecida.

Los datos de animación que el conversor exporta dependen del modelo y de las directivas que se hayan aplicado en MilkShape. Hay tres posibles situaciones, que el conversor EDL comprueba, exportando los datos de animación en consecuencia:

- Se exportan las transformaciones (traslación y rotación) de aquellas articulaciones que animen directamente una malla, es decir, que la malla no esté animada por otra articulación.
- Se exportan las transformaciones de aquellas articulaciones de las que se haya solicitado dichos datos expresamente mediante las directivas correspondientes. Estas transformaciones pueden ser necesarias para tener una referencia de la posición de un miembro del personaje a lo largo del tiempo.
- Se exportan los valores de vértices muestreados (y opcionalmente de normales) , transformados en cada muestra, de aquellos que son dependientes de diferentes articulaciones, por lo que no se pueden animar por transformación.

Las transformaciones muestreadas y exportadas no son jerárquicas (dependiendo la transformación de una articulación de la del "padre"), sino que son independientes, con el fin de poder animar un modelo mediante transformación y muestreo de vértices simultáneamente.

EDL permite almacenar hasta 255 esqueletos por modelo, por lo que se pueden realizar 255 animaciones independientes dentro de éste. Esto es útil para los escenarios por ejemplo: un modelo puede estar compuesto por una habitación y los objetos que hay en ella, incluyendo puertas y ventanas. Si se quieren animar estos objetos se les asigna un esqueleto distinto con animaciones independientes. De esta forma se puede ejecutar la animación de abrir y cerrar la puerta independientemente de la de abrir y cerrar la ventana. También se podría utilizar para animar las plataformas móviles en un escenario de un juego de plataformas en el que el personaje tiene que ir saltando de una a otra. De esta forma, se puede almacenar en un único modelo toda esta información de animación para objetos diferentes pero manejando solo un modelo.

En lo referente a los la información de los esqueletos, EDL contiene por una parte los datos de muestreo de animación de cada esqueleto. Éstos son:

- el **número de muestras** totales del clip de animación.
- el **periodo de muestreo** del clip de animación asociado a este esqueleto. Éste viene en formato con parte fraccionaria 12.4 para permitir más flexibilidad y facilidad a la hora de cambiar la velocidad de animación. Éste se empleará para ajustar la velocidad de la animación, teniendo en cuenta que la Nintendo DS va a 60 fps fijos.

Por otra parte, se almacena el **número de costumbres** de animación para poder manejar las distintas animaciones. Si este número es 0, significa que el esqueleto no está animado. Si es 1, no se incorpora la información relativa a las muestras finales de cada costumbre de animación, que se explica a continuación.

Con el fin de acotar cada costumbre de animación, EDL almacena la **muestra final** de cada una de forma sucesiva. Estos valores, en función de la costumbre de animación, se denominarán Ends(C). De esta forma, para la costumbre de animación C, las muestras que la componen estarán comprendidas en [Ends(C - 1) + 1, Ends(C)], excepto para la primera que será [0, Ends(0)]. Como ya se ha comentado, si el número de costumbres de animación es 1, esta información se obvia, dado que la muestra final se calcula como (Nº de muestras) – 1 y así se ahorra memoria.

Articulaciones

Además de los esqueletos, que aportan información de las muestras de animación y la velocidad a la que deben ser representadas, los datos de articulaciones aportan información de la posición y la rotación de éstas en cada muestra de la animación (si tiene), en el caso en el que se anime por transformación.

Además, a las articulaciones se les puede asociar un *billboard*. Un *billboard* es un objeto (un *sprite* o un plano), que normalmente porta una textura (animada o no), que siempre se orienta en dirección a la cámara, rotando según el movimiento de ésta. De esta forma, el *billboard* siempre está de frente a la cámara, dando la sensación de tridimensionalidad a partir de un plano. Hay varios tipos de *billboards*.

En este caso se han considerado dos tipos de *billboard*: el cilíndrico y el esférico. Ambos comparten la característica de que la dirección perpendicular a su plano apunta siempre en la dirección de la cámara, rotando en el eje vertical. Sin embargo, se diferencian en que el *billboard* cilíndrico no rota en el eje horizontal, de ahí su nombre, ya que su movimiento rotatorio dibuja un cilindro. El esférico, sin embargo, sí rota en el eje horizontal para apuntar a la posición de la cámara en todo momento. En la Figura 5.3 se ve de manera gráfica esta diferencia.

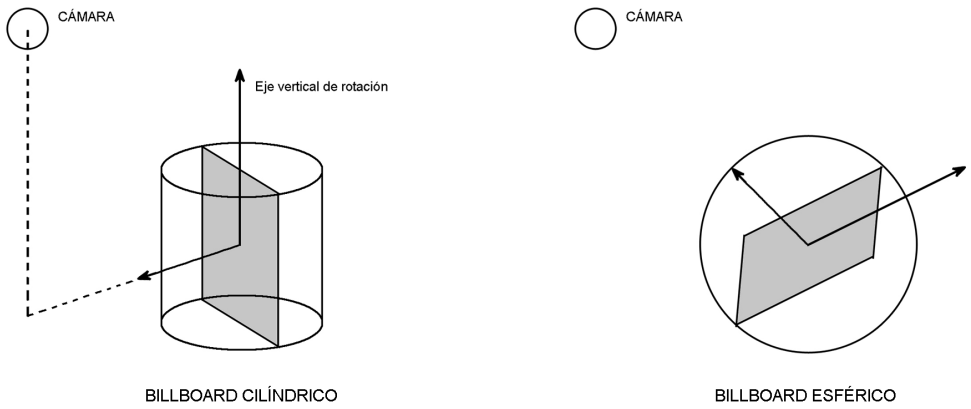

Figura 5.3: Billboards cilíndrico y esférico.

Una articulación con *billboard* tendrá restringido su movimiento al del *billboard* correspondiente, cilíndrico o esférico, restringiendo solo su rotación de muestra (aunque la mantiene). Dado que el sistema de transformación no es jerárquico, sólo se puede establecer

como *billboard* la articulación final de una rama del esqueleto, denominada articulación distal.

Entrando en materia de los datos de las articulaciones estructurados en el formato EDL, en primer lugar se almacena un **índice del esqueleto** al que pertenece la articulación, para poder asociar a está la animación que corresponda. Por otro lado, se establecen unas banderas que aportan información de animación:

- el estatus de ***billboard*** de la articulación y su **tipo**, cilíndrico o esférico.
- la presencia de **datos de traslación**
- la presencia de **datos de rotación**

En lo que se refiere a los propios datos de transformación, cada articulación incluye:

- una **traslación de reposo** (x, y, z), necesaria para las mallas que dependen únicamente de ésta articulación, con el fin de establecer la posición correcta de la articulación y la malla asociada en ausencia de animación (bien porque ésta no esté activa o bien porque la propia articulación no esté animada).
- **datos de traslación** (x, y, z) por cada muestra de la animación del esqueleto al que pertenece la articulación.
- **datos de rotación** (x, y, z, ángulo) por cada muestra de la animación del esqueleto al que pertenece la articulación.

Todos los datos de transformación vienen en formato f32, que es el que se emplea en las funciones de transformación más básicas de videoGL, ahorrando así operaciones de conversión en el renderizador.

Aunque una articulación no anime ninguna malla, se puede pedir la exportación de sus transformaciones, mediante directivas al conversor, para el equipado de un objeto externo. Por ejemplo, se pueden exportar las transformaciones de una articulación en el centro de una mano, que si bien no anima nada, puede servir para colocar un objeto en ésta y que acompañe su movimiento.

Mallas

En esta sección se incluyen los datos referentes a los vértices que componen cada una de las mallas del modelo.

Una cualidad especial en este apartado, y que es preciso explicar, es la inclusión de color por vértice. Ya se ha explicado que MilkShape 3D no permite esta opción, teniendo todos los vértices de la malla el mismo material. Sin embargo, este método de asignación de color es interesante, ya que se puede simular la apariencia de un modelo texturizado, prescindiendo de tal textura.

En este caso, se ha implementado dicho método a través del renderizador renderms y el conversor EDL. La aplicación renderms, que como ya se ha explicado, renderiza modelos en formato MilkShape ASCII, incorpora una funcionalidad con la cual se extrae el color de cada vértice independientemente, en función de la textura asociada a la malla y de su coordenada de textura. Así, a cada vértice del modelo se le asocia un color determinado en función de la textura asociada. Estos datos de color por vértice se exportan a un fichero con extensión .cpv, que el conversor procesará si es necesario, almacenando dichos datos en el fichero edl.

El color por vértice se puede emplear para objetos lejanos que no requieran el detalle de una textura, o bien para darle información visual adicional al modelo, como información de sombreado.

Por otra parte, hay que hacer un apunte sobre las coordenadas de textura y cómo las trata el conversor EDL. Éste realiza un análisis previo de las coordenadas de textura de cada malla, con el objetivo de deshacerse de todas las coordenadas de textura negativas, dado que la Nintendo DS no las soporta. Esto lo hace desplazando unidades enteras esta coordenadas de textura, de forma que todas ellas acaben siendo positivas.

Al igual que en los demás sectores del formato EDL, los datos de cada una de las mallas del modelo comienzan por las opciones y la información básica de éstas mediante banderas que indican:

- si el objeto es **sólido**. Si es así, en la etapa de render habrá que descartar las caras traseras (*backface culling*).
- si incluye **normales**, para procesarlas o no.
- si incluye **coordenadas de textura**, para procesarlas o no.
- si incluye **color por vértice**, para procesarlos o no
- si los **vértices** vienen en **formato V16** (16 bits por coordenada) o **V10** (10 bits por coordenada).
- si la **coordenada horizontal de textura** necesita **repetición**. De esta forma se repite en horizontal la textura sobre el modelo.
- si la **coordenada vertical de textura** necesita **repetición**. De esta forma se repite en vertical la textura sobre el modelo.
- si la malla está **animada**. Esto le indicará al renderizador si es preciso realizar cálculos de transformación sobre la malla.

Además de estas banderas, se proporciona información adicional necesaria para procesar la malla, en cuanto a su información de animación y el recorrido de sus vértices:

- **índice del esqueleto** que anima la malla. Si su valor es 255 la malla no está animada.
- **índice de la articulación** que anima la malla. Si su valor es 255, la malla no puede ser animada por transformación, solo por muestreo de vértices.
- **número de vértices** que componen la malla; hasta 1024 (10 bits).
- **número de normales** que componen la malla; hasta 1024 (10 bits). Este parámetro solo tiene información si hay normales.
- **índice del material** que le corresponde a la malla, de la lista de materiales almacenados a continuación.

A continuación, se almacenan los datos correspondientes a la información de los vértices que componen cada malla del modelo:

- las **coordenadas de los vértices** de la malla. Pueden exportarse en formato V10, es decir 10 bits por coordenada (en formato de punto fijo 4.6 con signo), ocupando 32 bits (los últimos dos bits quedan inutilizados), o bien en formato V16, es decir 16 bits por coordenada (en formato 4.12 con signo, ganando precisión), ocupando dos palabras de 32 bits, de los cuales los 16 últimos bits de la segunda palabra

(correspondiente a la coordenada z) quedan inutilizados.

- las **coordenadas de textura** correspondientes a cada uno de los vértices anteriores, solo en el caso de que éstas se incluyan. Son dos datos (u, v) de 16 bits cada uno en formato 12.4 (32 bits en total por coordenada de textura).
- el **color por vértice** correspondiente a cada uno de los vértices anteriores, solo en el caso de que se incluya esta opción. Se almacenan con 5 bits por componente rgb, por lo que en una palabra de 32 bits se almacenan dos colores por vértice (dejando un bit inutilizado de los 16 correspondientes a un color). En el caso de que haya un número impar de éstos, los últimos 16 bits se rellenan con ceros.
- las **normales** correspondientes a cada uno de los vértices anteriores, solo en el caso de que éstas se incluyan. Son tres datos en formato V10, es decir 10 bits por componente en formato 1.9 con signo.
- el **número de triángulos** que componen los vértices anteriores, para poder recorrer la lista de triángulos en la etapa de pintado y mandarle al hardware los vértices correspondientes a cada uno.
- los **índices a los vértices** que componen cada triángulo, para cada uno de los triángulos que forman la malla. De esta forma, cuando se recorren los triángulos en la etapa de pintado, se puede acceder a los vértices que los componen. Cada triángulo se almacena en una palabra de 32 bits, formada por los tres vértices que componen el triángulo, con 10 bits cada uno (1024 vértices máximo, es decir que el índice máximo es 1023).
- los **índices a las normales** correspondientes a los vértices que componen cada triángulo, para cada uno de los triángulos que forman la malla. De esta forma, cuando se recorren los triángulos en la etapa de pintado, se puede acceder a las normales que le corresponden. Se almacenan las tres normales correspondientes al triángulo en una palabra de 32 bits, con 10 bits cada una (1024 normales máximo, es decir que el índice máximo es 1023).

En el caso de que la malla esté animada pero ésta no pueda realizarse por transformación, se exportan los vértices y las normales (en el caso de que se incluyan) de cada muestra de la animación del esqueleto al que pertenece la malla en cuestión. Por lo tanto, se almacenan de nuevo los vértices (y opcionalmente las normales) de la malla con las coordenadas correspondientes a cada muestra de la animación, con el mismo formato que los vértices básicos almacenados anteriormente. Es decir que se almacenan los vértices correspondientes a la primera muestra de la animación, seguidos por los vértices correspondientes a la segunda muestra, y así sucesivamente. Por lo tanto el número de vértices adicionales V_m almacenados de esta manera será de

$$V_m = V \times M, \tag{5.1}$$

siendo V el número de vértices de la malla y M el número de muestras de la animación del esqueleto al que pertenece la malla.

De la misma forma, se almacenan las normales correspondientes a cada muestra de la animación, solo en el caso de que se incluyan éstas en los datos del modelo. Por lo tanto

se almacenarán N_m normales adicionales, cuyo valor viene dado por

$$N_m = N \times M, \tag{5.2}$$

siendo N el número de normales asociadas a la malla y M el número de muestras de la animación del esqueleto al que pertenece la malla.

Con estos datos es posible realizar el pintado de la geometría del modelo, quedando únicamente los datos de sus materiales para finalizar dicho proceso.

Materiales

Como última información del modelo, en el formato EDL se exportan los datos referentes a sus materiales. De esta forma, para cada uno de los materiales asociados al modelo se almacena su información, que consta de las opciones del material, las componentes del material (difuso, ambiente, emisivo y especular), las texturas asociadas (se pueden asociar dos texturas por material) e información de animación de texturas.

Aparte de la apariencia básica del material (componentes básicos y textura), al material en EDL se le puede asociar unas opciones adicionales poco empleadas en general en los proyectos 3D de *homebrew*. Éstas son la aplicación de una segunda textura y una capa adicional de reflexión esférica.

La segunda textura se procesa de la misma forma que la textura base, pudiendo modificar su transparencia de mezcla para visualizar ambas texturas mezcladas. También es posible aplicar una textura con canal de transparencia para que en aquellas zonas que sean transparentes se visualice la textura base y en las demás zonas se visualice la segunda textura.

La reflexión esférica consiste en una textura adicional, que no viene definida en el modelo, sino que se especifica externamente, dado que depende del entorno donde se encuentre el modelo, no del modelo en sí, que se aplica mediante unas coordenadas de textura calculadas a partir de las normales del modelo. De esta forma, se simula la reflexión del entorno sobre el modelo, otorgándole una apariencia pulida o metálica. Ésta capa también puede modularse con las capas inferiores (textura y segunda textura) mediante un parámetro de transparencia de mezcla.

Además, otra opción disponible es la de albergar datos de animación de textura por subimágenes. Ésta consiste en recorrer una imagen dividida en varias subimágenes en las que se representa un movimiento mediante la variación de una imagen a la siguiente. Es el mismo concepto que la animación tradicional, en la que el movimiento surge de la visualización sucesiva de imágenes con variaciones sutiles entre ellas. Por lo tanto, el recorrido se efectúa según el número de subimágenes en ancho y en alto de la textura, pasando de una a otra a una velocidad establecida por el periodo de la animación. De esta forma, se pueden conseguir animaciones simples de escenarios o de elementos que requieren poco detalle a partir de un plano con una textura animada, ahorrando geometría, así como efectos de partículas avanzados como fuego, humo, etc.

Las opciones del material están definidas en primer lugar por un parámetro de banderas que indican:

- si el material **tiene identificador de textura**, y por lo tanto textura asociada.

- si el material **tiene una segunda de textura** asociada.
- si el material **tiene la primera textura animada** (únicamente se puede animar la primera textura).
- si esta animación de textura se **activa cíclicamente** al cargar el modelo (puede interesar simplemente realizar un ciclo de animación en momentos determinados).

Además, las opciones del material vienen complementadas con parámetros adicionales referentes a parámetros de transparencia:

- la **transparencia de mezcla para la capa de reflexión**
- la **transparencia del material**, creando una apariencia translúcida en la malla
- la **transparencia de mezcla de la segunda textura**

Estos parámetros consisten en 5 bits, con valores entre 0 y 31, que es el formato empleado por la Nintendo DS para procesar la transparencia.

A continuación se almacenan los componentes difuso, ambiente, emisivo y especular del material, con 5 bits por componente rgb de cada color. De esta forma se emplean 15 bits por color (más uno inutilizado), agrupando los componentes difuso y ambiente en una palabra de 32 bits y el emisivo y el especular en otra.

Para asociar una textura al material, se almacena un identificador de hasta 12 caracteres con el cual buscar la textura por su nombre en la etapa de renderizado y aplicarla así al material a la hora de pintar el modelo. Cada carácter ocupa 1 byte, por lo que son necesarias 3 palabras de 32 bits para albergar este parámetro.

A continuación se almacenan los datos referentes a la animación por subimágenes, en el caso de que esté habilitada esta opción. En caso contrario se obvian dichos datos. Éstos son:

- el **periodo** de la animación en frames, almacenado en formato 12.4.
- el **ancho SIX** y el **alto SIY** en **subimágenes** de la textura, es decir cuántas subimágenes en alto y en ancho presenta la textura. De esta forma, el recorrido se realizará mediante traslaciones de (ancho de textura)/SIX en el eje x y (alto de textura)/SIY en el eje y según la subimágen que haya que visualizar en cada muestra de la animación.

Finalmente, al igual que en el caso de la textura base se almacena un identificador de hasta 12 caracteres para asociar al material dicha textura adicional en la fase de pintado y activación de las texturas del modelo, siempre que se haya habilitado la opción de la segunda textura. En caso contrario se obvian estos datos.

5.2.3. Versiones de EDL

A lo largo del desarrollo y a medida que se ha implementado la herramienta de render, se ha ido actualizando el conversor, y por lo tanto el formato EDL, incorporando nuevas opciones y funcionalidades en cada versión. Así, se han implementado cuatro versiones del formato, de la 0 hasta la 3, pudiendo elegir el formato de exportación en el conversor, generando éste un archivo con una estructura u otra en función de la versión indicada. A su vez, la herramienta de render, en la fase de carga del modelo EDL, comprueba la versión de éste y lee la estructura de datos en consecuencia. Por supuesto, lo lógico es

emplear la última versión, dado que es la más completa y la más depurada.

A continuación se enumeran, a grandes rasgos, las opciones limitadas disponibles en cada una de las versiones de EDL, teniendo en cuenta que la especificación previamente detallada es la correspondiente a la última versión, edl3.

Versión edl0

Esta es una versión preliminar muy limitada del formato EDL, en la que únicamente de incluye información de vértices y normales además de un material muy básico, sin tan siquiera posibilidad de asociar una textura. La animación no se contempla en este formato, por lo que los datos de esqueletos y articulaciones no existen (estos parámetros se incluyen en la última versión del formato).

En lo que se refiere a los datos de mallas, tan solo se incluyen las coordenadas de los vértices que las componen, con V10 como único formato soportado, las normales y los triángulos. Es decir, la información de geometría básica para empezar a pintar modelos en la Nintendo DS.

En cuanto a los materiales, éstos solo incluyen los componentes básicos de éste, es decir los colores difuso, ambiente, emisivo y especular.

Es por lo tanto un formato muy limitado, sin opciones de texturizado ni animación.

Versión edl1

La principal mejora de esta versión con respecto a edl0 es la inclusión de los datos de textura. Por lo tanto, en los datos de mallas se añaden las coordenadas de textura, mientras que en los de materiales se añade el parámetro del identificador de textura de 12 caracteres, además de los parámetros de banderas y opciones correspondientes en cada caso.

Adicionalmente, a los datos de materiales se añade el parámetro de transparencia del material y un parámetro correspondiente a la amplitud del resalte especular, posteriormente eliminado en el formato edl3, dado que no se emplea/no se puede modificar en la Nintendo DS.

Versión edl2

La tercera versión del formato EDL incorpora finalmente las coordenadas de vértices en formato V16, así como la opción del color por vértice, con las banderas y parámetros de control correspondientes.

5.3. Herramientas adicionales

Además del formato con el que cargar los modelos en la Nintendo DS, es necesario adaptar el formato de las imágenes que se usan como textura. Los formatos que se emplean para generar y procesar imágenes no son inteligibles por la Nintendo DS, por lo que es necesario exportarlas a un formato que sí lo sea.

La Nintendo DS maneja siete formatos de textura, cada uno con sus particularidades. Algunos de estos formatos requieren una paleta asociada. Las paletas son arrays que contiene el total de los colores disponibles para la textura. Por lo tanto, la textura no tiene información de color, sino índices a esta paleta, asociando así el color de cada texel (pixel de la textura). En la paleta, cada color está especificado por 15 bits, 5 bits por componente rgb. De esta forma, según el tamaño de la paleta (o el número de colores, es lo mismo) cada texel ocupará un número determinado de bits, según la profundidad que requiera la extensión de la paleta. Así, una textura con una paleta asociada de 4 colores, requiere una profundidad de bit por texel de 2 bits, para albergar cada uno de los 4 colores posibles de la paleta. De la misma forma, una textura con una paleta asociada de 256 colores requiere una profundidad de bit por texel de 8 bits, aumentando el tamaño de la textura en memoria.

Los formatos soportados por la Nintendo DS son:

- **Textura de 4 colores por paleta**: cada texel ocupa 2 bits.
- **Textura de 16 colores por paleta**: cada texel ocupa 4 bits.
- **Textura de 256 colores por paleta**: cada texel ocupa 8 bits.
- **Textura translúcida en formato A3I5**: formato que almacena información de transparencia. Cada texel ocupa 8 bits. Dispone de 3 bits para información de transparencia (A3), es decir, valores de 0 a 7, y 5 bits para el índice a una paleta de 32 colores (I5). La transparencia es expandida internamente para coincidir con el formato de 32 bits que maneja la Nintendo DS para transparencias, con la pérdida de precisión correspondiente.
- **Textura translúcida en formato A5I3**: formato que almacena información de transparencia. Cada texel ocupa 8 bits. Dispone de 5 bits para información de transparencia (A5), es decir, valores de 0 a 31, y 3 bits para el índice a una paleta de 8 colores (I3). La transparencia en este caso viene en el formato que maneja la consola.
- **Textura con color directo**: en este formato se almacena por cada texel el color real con 5 bits por componente rgb más 1 bit de transparencia (totalmente transparento u opaco), siendo éste el formato que maneja la Nintendo DS para los colores. De esta forma, cada texel ocupa 16 bits. Es por lo tanto un formato pesado.
- **Textura comprimida**: formato de textura comprimida que emplea un método complejo para almacenar bloques de 4x4 texels con paleta asociada. Dado que no se maneja este formato en este proyecto, no se entrará en detalle sobre su estructura y funcionamiento.

Dado que es necesario emplear estos formatos en el manejo de texturas, se requiere alguna herramienta que permita convertir los formatos de imagen estándar a éstos. Afortunadamente, en la escena *homebrew* se pueden encontrar diversas herramientas que facilitan el trabajo en este sentido. Éstas son grit [11] y Nitro Texture Converter [17].

5.3.1. Grit

Esta herramienta viene en el kit de desarrollo *homebrew* devkitPro. Ésta se emplea para convertir imágenes en formato de mapa de bits, bitmap, a cualquiera de los formatos con

paleta asociada. Sin embargo no convierte a los formatos con transparencia no trivial (que no sea 0 o 1).

La herramienta funciona de la siguiente manera: se meten las imágenes bitmap que se quieren convertir en una carpeta bitmap situada en el directorio raíz del programa; a continuación se ejecuta la herramienta, indicando ciertos parámetros de conversión, siendo el más importante, en este caso, la profundidad de bit del texel (asociado al número de colores de la paleta correspondiente).

El conversor genera dos archivos correspondientes a la imagen con los índices a los colores de la paleta y la propia paleta. Éstos son los archivos que se cargarán y se manejarán en la Nintendo DS.

5.3.2. Nitro Texture Converter

Dado que el conversor grit no exporta a los formatos con transparencia, ha sido necesario buscar una herramienta adicional que cumpla este cometido. Finalmente se ha hallado Nitro Texture Converter, un conversor que sí que aporta esta solución.

El conversor es capaz de convertir a los formatos A3I5 y A5I3, además de los formatos con paleta asociada RGB4, RGB16 y RGB256 y algún otro que no se emplea en el entorno de la Nintendo DS. Aunque esta herramienta convierta también a los formatos RGB4, RGB16 y RGB256, se ha utilizado grit para esta tarea ya que se comenzó con esta vía, además de que es la herramienta más extendida en la comunidad.

Nitro Texture Converter, a diferencia de grit, trabaja con el formato PNG, el cual dispone de un canal de transparencia. Por lo tanto, a partir de una imagen PNG, el conversor genera de nuevo dos archivos: el de imagen, que almacena el índice a los colores de la paleta y la transparencia de cada texel, y el de la paleta de colores que, dependiendo del formato especificado, será de 8 o 32 colores.

Capítulo 6

Librería de renderizado msNDS

Hasta ahora, se han expuesto y desarrollado las herramientas y la metodología necesarias para desarrollar aplicaciones (más concretamente, aplicaciones 3D) en la Nintendo DS. Además, se ha detallado la funcionalidad del formato específico de Nintendo DS para cargar modelos (EDL), así como su justificación en el marco de este proyecto.

De esta forma, se pasará a especificar y comentar la librería de funciones desarrollada íntegramente en este proyecto, msNDS, con el fin de dar abordar el objetivo del mismo: determinar las posibilidades del motor 3D de la Nintendo DS. Para ello, esta librería implementa un sistema de renderizado que aprovecha las capacidades gráficas de la consola.

Además, teniendo en cuenta que nos encontramos en el contexto de la escena *homebrew*, uno de los objetivos impuestos en el desarrollo de esta herramienta es que este sistema de render sea aprovechable por el resto de la comunidad, por lo que ésta ha de ser de fácil manejo y accesibilidad. Para ello, y como se detallará a continuación, en esta librería se contemplan dos entornos: el de usuario, con el que se maneja éste para realizar el pintado y la asignación de los estados del sistema, y el interno, al cual el usuario no tiene que acceder (ni debe hacerlo para evitar funcionamientos incorrectos), tratándose de elementos (tipos de datos y funciones) que son empleados por el entorno de usuario. Se ha dispuesto de esta forma para subdividir las diferentes tareas en el pintado, pero manteniendo el acceso a la tarea principal desde un único punto. De esta forma se consigue un código más claro y versátil.

Con esto, se pasa a especificar los dos módulos desarrollados para la librería: asset.h/c y edl.h/c. El primero se emplea para generar un sistema de almacenamiento de recursos (en este caso simplemente texturas) para ser utilizados en el pintado, y el segundo es el que se encarga de realizar el render de los modelos en formato EDL, leyendo, almacenando y procesando los datos de éstos. Por un lado se detallará la capa de acceso de cara al usuario, complementando con un pequeño manual de usuario que indique el modo de empleo de las funciones de usuario. Por otro lado, se detallará el funcionamiento interno del sistema de renderizado, explicando los tipos de datos y las funciones internas de éste. Se concluirá con un esquema de ejemplo de uso de la librería, dónde se detallarán las acciones a realizar para crear una aplicación 3D mediante el uso de ésta, de forma sencilla y sin necesidad de ahondar en las especificaciones de la Nintendo DS ni de lidiar con la capa intermedia que representa Libnds.

6.1. Asset.h

Este módulo de la librería msNDS facilita un sistema de gestión de recursos para, una vez almacenados en memoria principal, poder acceder a ellos fácilmente. En esta primera versión de la librería, el único tipo de recurso o *asset* es la textura, pudiendo por lo tanto manejar únicamente este tipo de elementos. El manejo de los recursos se efectúa mediante una lista de elementos, pudiendo insertar o eliminar elementos de ésta.

Para el manejo de texturas se ha definido una estructura que contiene su información, así como un recurso de tipo textura (conteniendo un puntero a la estructura anterior) que es el que manejará la lista de recursos.

Se han definido además una serie funciones con las cuales realizar operaciones sobre una lista y sus elementos, tales como iniciar la lista, definir un recurso, insertar un recurso en la lista, cargar una textura de la lista o eliminar una lista.

Estas estructuras y funciones se especifican a continuación.

6.1.1. Estructuras y tipos de datos

Para el correcto manejo de los recursos (en este caso únicamente texturas) se han definido una serie de tipos de datos estructurales correspondientes a las texturas, los recursos de tipo textura y las listas de recursos de tipo textura. Éstos son los que se enumeran a continuación.

texture_descriptor_t

- **const char * name**: nombre de la textura. Debe coincidir con el identificador de 12 caracteres almacenado en el archivo EDL para acceder a ésta y poder manejarla.
- **GL_TEXTUR_TYPE_ENUM texture_type**: tipo de textura, pudiendo ser RGB4, RGB16, RGB256, RGB (color directo), A5I3 y A3I5.
- **int tex_width**: ancho de la textura (GL_TEXTURE_SIZE_ENUM).
- **int tex_height**: alto de la textura (GL_TEXTURE_SIZE_ENUM).
- **int index0istransparent**: bandera que determina si ha de hacerse transparente el color del índice 0 de la paleta asociada a la textura.
- **const u8 * texture_data**: puntero a los datos de imagen de textura almacenados en memoria.
- **const u8 * palette_data**: puntero a los datos de paleta de textura almacenados en memoria.

Estructura que contiene la información relativa a una textura almacenada en memoria: su nombre, con el cual acceder a ella y que lo relaciona con el modelo, el tipo de textura, que debe ser uno de los manejados por la Nintendo DS (en esta primera versión las texturas comprimidas no están soportadas) sus dimensiones en vertical y horizontal, un indicador que determina si el color del índice 0 de la paleta ha de hacerse transparente, y los punteros a los datos de imagen y paleta asociada de la textura. Este tipo de datos de textura es además, como se verá más adelante, el que contiene la estructura definida

para los modelos.

El tipo de textura ha de especificarse mediante uno de los elementos del enumerador GL_TEXTURE_TYPE_ENUM definido en el módulo videoGL de Libnds, dado que se empleará en las llamadas a las funciones *glTexImage2d*() y *glTexParameter*(), ya comentadas en el capítulo referente a Libnds.

Igualmente, los parámetros width y height se han de especificar mediante uno de los elementos del enumerador GL_TEXTURE_SIZE_ENUM del módulo videoGL, dado que también se emplearán en las funciones *glTexImage2d*() y *glTexParameter*() y son éstos los valores esperados.

ASSET_TEXTURE

- **texture_descriptor_t *pTextureDescriptor**: puntero a la textura correspondiente al recurso.
- **int textureID**: identificador generado por *glGenTextures*() y asignado a la textura correspondiente al recurso.
- **int assetID**: identificador numérico que define la posición del recurso en la lista de recursos en la que se ha insertado.

Esta estructura es necesaria para crear y manipular recursos de tipo textura, en los cuales se contiene la dirección de la textura cargada en memoria principal. Estos elementos se crean mediante la función *SetTextureAsset*() de este mismo módulo y se insertan en la lista principal de recursos de tipo textura o en cualquiera que se haya creado mediante *AddTextureAssetToCurrentAssetList*() o *AddTextureAsset*() respectivamente.

Los parámetros *textureID* y *assetID* no se usan en el sistema EDL, ya que el primero se almacena en la propia estructura edl, resultando más cómodo su acceso, y las texturas se buscan por nombre. Se han mantenido porque pueden resultar útiles ante la implementación de un sistema de carga de modelos diferente al EDL.

ASSETLIST_TEXTURE

- **ASSET_TEXTURE *pFirstAsset**: puntero al primer recurso contenido en la lista.
- **ASSET_TEXTURE *pCurrentAsset**: puntero al recurso actual de la lista. Este parámetro sirve para desplazarse por la lista y analizar su contenido.
- **int maxAssets**: número máximo de recursos que la lista puede contener.
- **int numAssets**: número de recursos contenidos hasta el momento en la lista.
- **int *textureIDs**: puntero al array dónde se almacenan los identificadores generados por *glGenTextures*(), necesarios para activar las texturas correspondientes a cada recurso.

Esta estructura define un elemento lista de recursos de tipo textura en el cual contener las texturas que se deseen. Al iniciar o crear una lista se define su tamaño con el número máximo de texturas que ha de contener. Esto se hace mediante las funciones *InitMainAssetList*() (inicia la lista principal definida por defecto) o *CreateTextureAssetList*() (se crea una lista diferente a la principal, en el caso en el que se requieran dos listas independien-

tes). Para cambiar la lista con la que se va a operar se utiliza *SetCurrentAssetlistTexture*() y para eliminar listas se llama a las funciones *DestroyCurrentTextureAssetList*() o *DestroyTextureAssetList*().

La búsqueda de texturas en una lista de recursos se efectúa comparando el nombre de la textura del modelo y el de la textura del recurso. De esta forma se recorre la lista hasta encontrar la el recurso de tipo textura especificado (si es que éste está contenido en la lista). Esta operación se realiza llamando a la función *load_texture*().

El parámetro *textureIDs* no se usa en el sistema EDL, ya que en este sistema cada identificador se asigna al material del modelo edl. Se ha mantenido porque puede resultar útil ante la implementación de un sistema de carga de modelos diferente al EDL.

ASSETLIST_TEXTURE *MAIN_ASSETLIST_TEXTURE

Esta es una variable global de tipo puntero a una lista de recursos de tipo textura que apunta a la lista principal definida por defecto. Si no se crea ninguna otra lista de recursos ésta es la que se utilizará para gestionar y manejar las texturas de los modelos. Ha de iniciarse con *InitMainAssetList*().

ASSETLIST_TEXTURE *CURRENT_ASSETLIST_TEXTURE

En el caso en el que se cree una lista de texturas adicional a la principal, se da la opción de definirla como lista actual o activa, de forma que se pueden emplear funciones específicas en las que se trabaja con este parámetro y así no tener que definir en cada momento con qué lista se está trabajando.

Hay funciones que internamente trabajan directamente con este parámetro, por lo que es imprescindible tenerlo actualizado, de forma que se acceda a la lista adecuada en todo momento. Por ejemplo, en el sistema de render, la carga de texturas de un modelo se realiza accediendo a la lista activa, para no tener que pedir la lista con la que trabajar. Por ello, antes de pintar un modelo, es necesario especificar la lista activa (siempre que se trabaje con listas adicionales a la principal). Por esta misma razón, es necesario que todas las texturas de un mismo modelo estén contenidas en la misma lista.

6.1.2. Funciones

Para gestionar los recursos y las listas que los contienen, se da a disposición en el módulo asset una serie de funciones tanto de especificación como de acceso. Éstas se detallan a continuación.

void InitMainAssetList(int maxAssets,int *textureIDs)

Con esta función se inicializa la lista principal de recursos de tipo textura, es decir, la que se define por defecto. En ésta se reserva la memoria necesaria para contener un número máximo de texturas *maxAssets*, definiendo así su tamaño, y se inicializan sus parámetros. La dirección de memoria de esta lista está contenida en la variable global MAIN_ASSETLIST_TEXTURE, por lo que se puede acceder a dicha lista sin necesidad de

que el usuario aporte este dato. En la inicialización CURRENT_ASSETLIST_TEXTURE se iguala a MAIN_ASSETLIST_TEXTURE para, en el caso de no crear ninguna otra lista, sea por defecto ésta la que se utilice para cargar las texturas en el sistema de renderizado.

Es necesario llamar a esta función para poder utilizar la lista principal de recursos, y por lo tanto acceder correctamente a las texturas en cada momento.

Dado que, cómo se ha explicado, el parámetro *textureIDs* de la estructura ASSETLIST_TEXTURE no se utiliza en el sistema EDL, el parámetro de entrada de esta función, *textureIDs*, tampoco es necesario, por lo que puede ser 0 (o cualquier otro valor).

ASSETLIST_TEXTURE* CreateTextureAssetList(int maxAssets, int *textureIDs)

En el caso en el que se requiera tener texturas agrupadas en diferentes listas, se dispone de esta función, con la que se crea e inicializa (de forma análoga a la lista principal) una lista adicional de recursos de tipo textura. Ésta devuelve el puntero a dicha lista.

A diferencia de *InitMainAssetList*(), en esta función no se asigna la dirección apuntada por el puntero devuelto a CURRENT_ASSETLIST_TEXTURE, por lo que se mantendrá la lista principal como la activa (en el caso de que se haya inicializado) hasta que se cambie de forma manual llamando a la función *SetCurrentAssetlistTexture*().

void SetCurrentAssetlistTexture(ASSETLIST_TEXTURE *pAssetlistToMain)

Esta función establece la lista pasada como parámetro como la lista de recursos de tipo textura activa. De esta forma se accederá a los recursos contenidos en dicha lista en el momento de cargar las texturas. Hay que tener cuidado con esto, ya que si se cambia la lista activa y se quieren cargar las texturas de un modelo que no estén contenidas en ésta, no se encontrarán y por lo tanto no se pintarán.

ASSET_TEXTURE *SetTextureAsset(texture_descriptor_t *texDescriptor)

Esta función ha de emplearse para establecer un recurso de tipo textura a partir de un elemento de tipo *texture_descriptor_t* correspondiente a una textura. Éste será asignado al parámetro correspondiente de la estructura ASSET_TEXTURE, pudiendo así introducirlo en una lista de recursos.

int GetTexWidth(texture_descriptor_t *texDescriptor)

Dado que el parámetro *tex_width* de la estructura *texture_descriptor_t* no corresponde al ancho en píxels de la textura, sino a un identificador empleado por Libnds (ver documentación), se ha puesto a disposición esta función de acceso para conocer dicha información.

int GetTexHeight(texture_descriptor_t *texDescriptor)

Dado que el parámetro *tex_height* de la estructura *texture_descriptor_t* no corresponde al alto en píxels de la textura, sino a un identificador empleado por Libnds (ver documentación), se ha puesto a disposición esta función de acceso para conocer dicha información.

void AddTextureAssetToCurrentAssetList(ASSET_TEXTURE* asset)

Con esta función se introduce un recurso de tipo textura en la lista activa, entregando como parámetro de entrada la dirección de memoria (o el puntero) de dicho recurso. Al añadir un elemento a la lista, se actualizan los parámetros *pCurrentAsset* y *numAssets*. Si se ha alcanzado el número máximo de texturas que puede contener la lista, el recurso no se inserta.

void AddTextureAsset(ASSETLIST_TEXTURE *assetList, ASSET_TEXTURE *asset)

Con esta función se introduce un recurso de tipo textura en la lista especificada con el parámetro de entrada/salida *assetList*, entregando como parámetro de entrada la dirección de memoria (o el puntero) de dicho recurso. Al añadir un elemento a la lista, se actualizan los parámetros *pCurrentAsset* y *numAssets*. Si se ha alcanzado el número máximo de texturas que puede contener la lista, el recurso no se inserta.

void DestroyCurrentTextureAssetList()

Con esta función se elimina la lista actual de recursos de tipo textura, libreando el espacio de memoria previamente reservado. La variable global CURRENT_ASSETLIST_TEXTURE se iguala a NULL para evitar que apunte a cualquier otra dirección y pudiendo causar errores.

void DestroyTextureAssetList(ASSETLIST_TEXTURE *assetList)

Con esta función se elimina la lista recursos de tipo textura especificada por el parámetro de entrada/salida *assetList*, libreando el espacio de memoria previamente reservado. En el caso de que esta lista fuera la activa, la variable global CURRENT_ASSETLIST_TEXTURE se iguala a NULL para evitar que apunte a cualquier otra dirección y pudiendo causar errores.

texture_descriptor_t *load_texture(char *name)

Esta función realiza la búsqueda de una textura en la lista activa de recursos de tipo textura. La búsqueda se realiza recorriendo la lista y comparando el nombre del recurso proporcionado (*name*) con el de cada uno de los elementos contenidos en ésta. Si la textura es encontrada, la función devuelve el puntero a dicha textura.

Esta función se emplea en el sistema de renderizado a la hora de cargar las texturas de un modelo en la memoria de vídeo.

6.2. Edl.h: capa de acceso y manual de usuario

El módulo edl es el que contiene el sistema de renderizado basado en el formato Extended Display List, específicamente diseñado para la Nintendo DS. En él se han implementado las funciones que realizan todas las operaciones necesarias para pintar un modelo en formato EDL, atendiendo a sus propias características y las propiedades que el usuario desee en cada momento. De esta forma, a través de unas pocas funciones se ejecuta un complejo sistema de pintado que resulta invisible de cara al usuario. Por lo tanto, se simplifica mucho la tarea de desarrollo al usuario en este contexto, aportando además un destacable control sobre el renderizado y la animación.

Este sistema se basa en la librería Libnds, que a su vez ejerce de interfaz con el hardware de la consola, si bien en algunos casos es necesario bajar a una capa de abstracción menor. Por lo tanto, se trata de una capa intermedia entre el usuario y Libnds, por lo que cualquier usuario con pocos conocimientos, tanto de programación como del funcionamiento de la Nintendo DS, tiene acceso a las capacidades gráficas de la consola a través del sistema de renderizado implementado.

En este módulo se ponen a disposición del usuario una serie de tipos de datos y de funciones que activan y controlan el sistema de pintado. Estos son los elementos con los que el usuario de la librería debe trabajar, con el fin de que el sistema funcione correctamente.

A parte de la referencia a los tipos de datos y las funciones propias de la capa de acceso, se establecerá un pequeño manual de usuario que servirá como guía de utilización de la librería, donde se describirán los pasos a seguir para configurar y lanzar el sistema de renderizado y de animación.

6.2.1. Estructuras y tipos de datos de usuario

En esta sección se detallan las estructuras y tipos de datos que los usuarios de la librería deben usar para poner en marcha el sistema de renderizado implementado. Es importante emplear únicamente éstos ya que el sistema está diseñado a tal efecto, y de lo contrario se podrían producir errores e inestabilidades en el mismo.

De la misma forma, es importante no escribir sobre estos datos, es decir variar directamente parámetros de estas estructuras. Para ello, se han de emplear las funciones de acceso que se ponen a disposición en la librería.

Los tipos de datos centrales de la librería son los correspondientes a los modelos edl y sus instancias, necesarias para pintar varios elementos de un mismo modelo con estados de animación independientes. Por lo tanto, una instancia de modelo edl tiene un parámetro que almacena la dirección del modelo edl del cual es instancia, que a su vez está compuesto por listas de elementos de tipo esqueleto, articulación, malla y material. En la Figura 6.1 se muestra un esquema que ilustra la organización de los datos en dichas estructuras.

A continuación se enumeran las estructuras y tipos de datos pertenecientes a la capa de acceso al sistema de renderizado que ha de emplear el usuario.

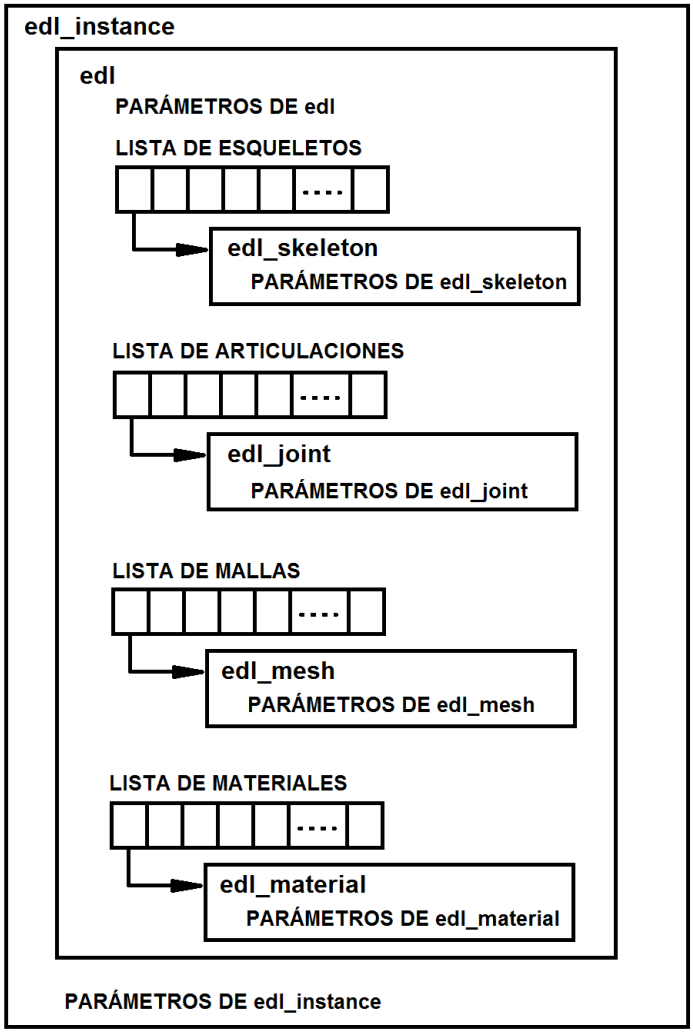

Figura 6.1: Organización de las estructuras que componen el elemento de un modelo edl.

edl

- **unsigned int * data**: puntero a los datos del archivo EDL cargado en memoria.
- **int nmeshes**: número de mallas que componen el modelo.
- **int nmaterials**: número de materiales asociados al modelo.
- **int nskeletons**: número de esqueletos asociados al modelo.
- **int njoints**: número de articulaciones asociadas al modelo.
- **int flags**: banderas que determinan ciertas propiedades del modelo. (Ver especificación EDL).
- **edl_mesh * meses**: puntero al inicio de la lista de mallas del modelo.
- **edl_material * materials**: puntero al inicio de la lista de materiales del modelo.
- **edl_skeleton * skeletons**: puntero al inicio de la lista de esqueletos del modelo.
- **edl_joint * joints**: puntero al inicio de la lista de articulaciones del modelo.

Esta estructura contiene los datos y las propiedades de un modelo EDL. Es por lo tanto

uno de los elementos centrales del sistema de renderizado. En ésta se agrupan todos los elementos de tipo malla, material, esqueleto y articulación que lo componen, así como sus parámetros de control y sus propiedades en forma de banderas.

Cada elemento malla, material, esqueleto o articulación no se almacena en la propia estructura, sino que se almacena un puntero que apunta a la dirección de memoria del inicio de las sucesiones de tales elementos en los datos del modelo cargados en memoria (accesibles mediante el parámetro data). Por lo tanto, conociendo el inicio de los datos y el número de éstos, es posible procesarlos con total control. Por ejemplo, en el renderizado del modelo, se recorren sus mallas para pintarlas. En este caso basta con acceder a la dirección de memoria del inicio de la lista de éstas e ir avanzando con pasos del tamaño del tipo *edl_mesh*, procesando en cada caso la malla correspondiente.

El parámetro *flags* contiene las banderas que aportan información del modelo en relación a:

- si el modelo tiene algún **material con reflexión**
- si el modelo tiene algún **material con segunda textura**
- el **orden de pintado de las capas de reflexión y de segunda textura** (esto depende de si se desea que a la segunda textura se le aplique también el efecto de reflexión o no)
- si el modelo tiene **texturas**
- si el modelo tiene **datos de animación**
- si el modelo tiene algún **material** asociado
- si el modelo aporta información de **normales**
- si el modelo tiene algún **material con animación de textura**
- si alguna articulación del modelo está afectada por un **billboard**

Estos estados se controlan aplicando a este parámetro una máscara de bit cuyo valor se extrae del enumerador *model_flags* (estructura interna de control), conteniendo todas estas posibilidades. De esta forma se pueden conocer todas estas propiedades, para tener un control total sobre el estado del modelo. Este proceso de escaneo de los atributos del modelo se efectúa de manera interna, por lo que el usuario no requiere realizar esta operación, sino que puede acceder a ellas a través de las funciones de acceso puestas a disposición.

edl_instance

- **edl *edlp**: puntero al modelo EDL del cual es instancia este elemento. Contiene así la información de apariencia del modelo.
- **float * anim_speed_rate**: puntero a un array que contiene el coeficiente de velocidad de la animación de cada esqueleto de la instancia del modelo, para variaciones de velocidad. Se trata de un multiplicador que modifica el periodo de muestreo de la animación, acelerándola o ralentizándola. El valor por defecto es 1 (corresponde al 100 % de velocidad), no modificando la velocidad. Valores mayores aumentan la velocidad de la animación, menores la reducen.
- **float * current_skeleton_sample**: puntero a un array que contiene la muestra

actual de cada esqueleto de la instancia del modelo.

- **int * frame_counter**: puntero a un array que contiene un contador de frames correspondiente a cada esqueleto de la instancia del modelo.
- **int * current_skeleton_anim**: puntero a un array que contiene la costumbre de animación actual de cada esqueleto de la instancia del modelo.
- **int * skeleton_anim_cyclic**: puntero a un array que contiene el estado cíclico actual de la animación de cada esqueleto de la instancia del modelo.
- **int * skeleton_anim_repetitions**: puntero a un array que contiene el número de repeticiones de animación restantes que reproducir para cada esqueleto de la instancia del modelo.
- **int * skeleton_anim_playing**: puntero a un array que contiene el estado de reproducción de la animación de cada esqueleto de la instancia del modelo.
- **animation_mode * anim_mode**: puntero a un array que contiene el modo de animación actual de cada esqueleto de la instancia del modelo.
- **int * ping_pong_state**: puntero a un array que contiene el estado del modo ping-pong (adelante o atrás) de la animación de cada esqueleto de la instancia del modelo (en caso de estar en modo de animación ping-pong).
- **int n_anim_tex**: número de texturas animadas del modelo.
- **int * anim_tex_material_id**: puntero a un array que contiene los índices a los materiales del modelo con animación de texturas.
- **int * anim_tex_xPos**: puntero a un array que contiene la posición en horizontal de la subimagen actual de cada material animado de la instancia del modelo.
- **int * anim_tex_yPos**: puntero a un array que contiene la posición en vertical de la subimagen actual de cada material animado de la instancia del modelo.
- **float * anim_tex_current_frame**: puntero a un array que contiene el frame actual de cada material animado de la instancia del modelo.
- **float anim_tex_speed_rate**: puntero a un array que contiene el coeficiente de velocidad de la animación de textura de cada esqueleto de la instancia del modelo, para variaciones de velocidad. Se trata de un multiplicador que modifica el periodo de muestreo de la animación de textura, acelerándola o ralentizándola. El valor por defecto es 1 (corresponde al 100 % de velocidad), no modificando la velocidad. Valores mayores aumentan la velocidad de la animación, menores la reducen.

Esta estructura puede decirse que es el elemento central del sistema de renderizado implementado en la librería msNDS. Representa una instancia o un elemento de un modelo EDL, con sus estados de animación independizados. En ésta se almacena, además de la dirección de memoria de los datos del modelo EDL en el cual se basa (que aporta su información de geometría, de muestreo y de apariencia), datos de animación para cada esqueleto y material de textura, independientes entre sí, e independientes de las demás instancias del mismo modelo.

Así, para cada esqueleto, se almacena su estado de animación que depende tanto del tiempo (que determina el valor del contador de frames y de la muestra de animación actual según la costumbre de animación activa), como del modo de animación seleccionado por el usuario (estos pueden ser *forward*, hacia adelante, *backward*, hacia atrás, y *ping-pong*,

en ambos sentidos de forma consecutiva). De tal forma, al pintar el modelo se tienen en cuenta los datos de animación propios de cada esqueleto, por lo que quedan totalmente independizados, tanto entre esqueletos del mismo modelo, como entre esqueletos de diferentes instancias del mismo modelo.

Esto confiere al sistema un control sobre la animación considerablemente alto, permitiendo al usuario un gran margen de maniobra, tanto a la hora de emplazar y animar varias instancias de un mismo modelo, como a la hora de animar de forma independiente cada uno de los esqueletos de un modelo. Este control está reforzado por las funciones de activación y configuración de animaciones, pudiendo en todo momento pasar de animar cíclicamente una costumbre de animación, a animar un número determinado de repeticiones otra costumbre, y pudiendo cambiar en cualquier momento el modo de animación (como se explicará más detalladamente en el apartado dedicado a las funciones del módulo edl).

Éste es uno de los puntos fuertes del sistema de render implementado, ya que si bien se ven en la escena *homebrew* varias herramientas de pintado 3D, éstas no aportan prácticamente nunca sistemas de animación completos, en el caso en el que ésta esté siquiera contemplada (que rara vez lo hacen).

animation_mode

- **ANIM_FORWARD**
- **ANIM_BACKWARD**
- **ANIM_PINGPONG**

Este enumerador contiene las definiciones de los diferentes modos de animación que se pueden aplicar. Éstos se aplican llamando a la función *set_anim_play_mode*(), a la que se le pasa la instancia del modelo afectada, el índice del esqueleto que se desee modificar y por último el modo de animación que se requiera.

edli_translation

- **unsigned int x**
- **unsigned int y**
- **unsigned int z**

Esta estructura alberga datos de traslación en el formato en el que se almacenan las mismas en el modelo edl. Ésta se emplea para recibir las traslaciones exportadas de una articulación, mediante la llamada a la función *get_current_joint_translation*(), a la que se le pasa la instancia del modelo considerada y la articulación de la que se desea extraer su traslación, devolviendo la traslación correspondiente a la muestra actual de la animación.

edli_rotation

- **unsigned int x**
- **unsigned int y**
- **unsigned int z**

- **unsigned int angle**

Esta estructura alberga datos de rotación en el formato en el que se almacenan las mismas en el modelo edl. Ésta se emplea para recibir las rotaciones exportadas de una articulación, mediante la llamada a la función *get_current_joint_rotation*(), a la que se le pasa la instancia del modelo considerada y la articulación de la que se desea extraer su rotación, devolviendo la rotación correspondiente a la muestra actual de la animación.

rendering_options

- **ro_use_normals = 1**
- **ro_use_texcoords = 2**
- **ro_use_texture = 4**
- **ro_use_material = 8**
- **ro_use_cpv = 16**
- **ro_use_scnd_texture = 32**
- **ro_use_reflection = 64**
- **ro_animate = 128**
- **ro_use_billboard = 256**
- **ro_use_everything = 0xFFFF**

Esta enumeración contiene las diferentes opciones de renderizado que se pueden aplicar en el momento del pintado de un modelo. Éstas se activan y se comprueban mediante operaciones y máscaras de bit respectivamente. Por lo tanto, el proceso para configurar las opciones de renderizado que han de aplicarse al pintado de un modelo es el siguiente: se define una variable entera que será el parámetro de las opciones de renderizado, se le asigna el valor ro_use_everything, que pone todos los bits a 1 (lo que corresponde a todas las opciones activas), en el caso en el que se requieran todas las opciones al inicio; en el caso en el que se desee activar una opción determinada, se realiza una operación de bit OR con el valor de la opción determinada sobre esta variable, activando la opción deseada; en el caso en el que se quiera invertir el estado de una opción, se realiza una operación XOR con el valor de la opción determinada sobre esta variable, activando o desactivando la opción deseada según el estado previo.

model_properties

- **EDL_HAS_NORMALS**
- **EDL_HAS_TEXTURES**
- **EDL_HAS_ANIMATION**
- **EDL_HAS_CPV**
- **EDL_HAS_MATERIALS**
- **EDL_HAS_REFLECTION**
- **EDL_HAS_TEXTURE_ANIMATION**

- **EDL_HAS_SCND_TEXTURE**
- **EDL_HAS_BILLBOARD**
- **EDL_REFLECTION_SCND_TEX_ORDER**

Este enumerador alberga todas las posibles propiedades que puede contener un modelo edl. Estos parámetros se emplean para obtener información relativa a los atributos del modelo pasándolos a la función *has_model_propertie*(), que devuelve 1 en caso de que el modelo disponga de la característica especificada y 0 en caso contrario.

6.2.2. Funciones de usuario

Finalmente, se ha puesto a disposición una serie de funciones de usuario, con las que acceder a todas las funcionalidades del sistema de render, así como a los parámetros de los modelos creados en la aplicación 3D. Éstas son las únicas que el usuario debe emplear para mantener el buen funcionamiento del sistema y evitar errores e inestabilidades.

edl * build_edl(unsigned int * data)

Esta función lee el modelo almacenado en un fichero EDL, cuya dirección de los datos cargados en memoria viene definida por data. Ésta, internamente, verifica la versión del fichero EDL y llama a la función correspondiente en cada caso: *build_edl0*(), *build_edl1*(), *build_edl2*() o *build_edl3*(). De esta forma, para el usuario esta operación es transparente.

Por lo tanto, esta función reserva la memoria necesaria para albergar los datos del modelo y carga aquellos que estén soportados por la versión correspondiente en la estructura edl, devolviendo el puntero a dicho modelo.

Una vez creado este modelo, se pueden instanciar tantos elementos de éste como se precise (siempre teniendo en cuenta las limitaciones de memoria de la consola).

int delete_edl(edl * edlp)

Esta función destruye un modelo edl, libreando la memoria que se ha reservado para mallas, materiales, articulaciones, esqueletos y el propio modelo, en el momento de la creación del modelo, es decir, en la función de lectura de éste.

edl_instance * get_edl_instance(edl * edlp)

Esta función crea una instancia del modelo edl especificado, inicializando los parámetros de animación que éste requiera. De esta forma, dependiendo del número de esqueletos de los que disponga, se reserva memoria para albergar:

- las **muestras de animación** actuales
- los **contadores de frames**
- las **costumbres de animación** actuales
- los **estados de reproducción**
- los **modos** de animación

- los **estados de animación cíclica**
- el **número de repeticiones** de reproducción de animación
- los **estados del modo ping-pong**
- las **velocidades de animación**

De esta forma, se reserva espacio para controlar la animación de cada esqueleto de forma independiente.

Igualmente, se reserva espacio en memoria para almacenar los datos de animación de textura, atendiendo al número de materiales con dicha propiedad, para almacenar:

- los **identificadores** de los materiales con **textura animada**
- las **posiciones en horizontal** de las subimágenes actuales
- las **posiciones en vertical** de las subimágenes actuales
- los **frames actuales** de animación
- las **velocidades de animación de textura**

Adicionalmente, se inicializa a 0 los parámetros numéricos correspondientes a cada esqueleto, a 1 los coeficientes de velocidad de animación y de animación de textura de cada esqueleto y material animado respectivamente, correspondiendo con la velocidad por defecto, y a ANIM_FORWRD el modo de animación de cada esqueleto.

void destroy_edl_instance(edl_instance * edli)

Esta función destruye una instancia de un modelo edl, liberando la memoria reservada para los datos de animación en el momento de su creación.

void render_edl(edl_instance * edl_instance, int options, u32 poly_attr)

Esta función pone en marcha el sistema de renderizado para la instancia de un modelo edl especificada, aplicando las opciones de renderizado y unos atributos de polígono primarios (comunes a todas las mallas del modelo). Estos últimos corresponden principalmente a las luces de la escena que han de afectar al modelo, especificándolo con los parámetros POLY_FORMAT_LIGHT (de 0 a 3, ya que se pueden definir hasta 4 luces). El resto de atributos se extraerán internamente según las propiedades específicas del modelo y sus elementos.

Internamente, esta función realiza una llamada a render_pass() por cada pasada que requiera el modelo, según disponga de un material con segunda textura o de reflexión, atendiendo al orden de pintado de éstas.

Además, si el modelo está animado, sus datos de animación, tanto de geometría como de textura son actualizados para el frame siguiente mediante la llamada a las funciones internas *edli_next_sample*() y *edli_tex_anim_next_sample*() respectivamente.

void load_model_textures(edl * edlp)

Esta función carga las texturas de un modelo edl de la lista de recursos activa llamando a la función *load_texture*() del módulo asset. De esta forma se asignan dichas texturas al

parámetro *texture_descriptor* de cada uno de los materiales que dispongan de ésta.

Esta función ha de llamarse antes de pintar las instancias del modelo edl en cuestión, para el correcto acceso a éstas. Debe realizarse fuera del bucle principal, de lo contrario en cada ciclo se realizaría esta operación, hecho que consumiría tiempo de procesado en una operación innecesaria (ya que una vez cargadas las texturas de un modelo en memoria de video, éstas pueden utilizarse indefinidamente).

void free_model_textures(edl * edlp)

Esta función desasigna la textura del parámetro *texture_descriptor* de cada uno de los materiales del modelo, eliminando dicha información de los datos de éste. Esta operación es independiente de la eliminación de las texturas del modelo de la memoria de video, por lo que es necesario realizar ambas operaciones cuando se vayan a dejar de emplear dichas texturas.

void convert_texture_coordinates(edl * edlp, int forward)

Esta función convierte las coordenadas de textura del modelo edl, previamente cargado con *build_edl*(), a un formato y rango adaptados a las dimensiones de la textura, debido a que dichas coordenadas no vienen en formato de punto fijo 12.4 propio de las coordenadas de textura en la Nintendo DS, variando los valores previamente almacenados en el modelo.

Las coordenadas de textura en la Nintendo DS no tienen un formato normalizado (como en openGL), sino que tienen un formato dependiente del tamaño de la textura. En el proceso de conversión del modelo al formato EDL desde MilkShape ASCII no se conoce el tamaño de la textura (únicamente se conocen los nombres de éstas), por lo que se insertan en un formato normalizado que ha de ser convertido a posteriori mediante esta función. Además, de esta forma, es posible cambiar el tamaño de la textura sin afectar al modelo convertido.

Esta función ha de llamarse después de haber cargado el modelo edl y sus texturas, con el sentido de conversión hacia delante (pasando 1 como parámetro *forward*) para transformar correctamente sus coordenadas de textura antes de ser aplicadas. Para realizar la operación inversa se le pasa a la función como parámetro *forward* el valor 0.

void bind_model_textures(edl * edlp)

Esta función genera las texturas correspondientes a un modelo edl en el hardware mediante la llamada a *glGenTextures*(), cargándolas en memoria de video y posibilitando la activación de éstas en el momento del pintado (ver Figura 6.2). Una vez generada la textura, se le asignan los parámetros que le correspondan, de acuerdo al formato de textura, sus dimensiones, si requiere repetición en horizontal o en vertical o si hay que establecer el color de índice 0 transparente. Estas propiedades se van acumulando en un parámetro entero, para aplicarlos en una sola llamada a *glTexParameter*(). De hacer varias asignaciones con esta función, se sobrescribirían los parámetros previamente establecidos, resultando en una configuración de la textura errónea.

Esta función ha de llamarse antes de pintar las instancias del modelo edl en cuestión, para la correcta activación de sus texturas. Debe realizarse fuera del bucle principal,

de lo contrario en cada ciclo se realizaría esta operación, hecho que consumiría tiempo de procesado en una operación innecesaria (ya que una vez cargadas las texturas de un modelo en memoria de video, éstas pueden utilizarse indefinidamente).

Para cargar las texturas en memoria y activarlas, se sigue el proceso siguiente: se buscan las texturas del material llamando a la función del módulo asset *load_texture*() (función interna) pasándole el parámetro *asset_id* (o *asset_scnd_tex_id* en el caso de la segunda textura); esta llamada se realiza en el módulo edl mediante la función de usuario *load_model_textures*(); en el caso en el que se encuentren, se asigna al parámetro *texture_descriptor* del modelo EDL la dirección de la textura, contenida en el recurso; con este dato, en el momento de activar las texturas con la función *bind_model_textures*(), se genera la textura llamando a *glGenTextures*(), pasándole la dirección de *texture_id* (o *scnd_texture_id*), al que se asocia el identificador de textura generado. Con este dato se puede activar la textura con *glBindTexture*() para que sea pintada. Este flujo de datos en las funciones de carga y activación de texturas se ve más claramente en la Figura 6.2 (en este esquema solo se muestran las operaciones sobre los datos de textura, no el proceso interno integral de las funciones, dado que se realizan más operaciones a parte de las mostradas; para ello ver el código).

void unbind_model_textures(edl * edlp)

Esta función elimina las texturas de un modelo de la memoria de vídeo, liberando esta memoria para la carga de otras texturas. Es importante realizar esta operación cada vez que se va a dejar de pintar un modelo, con el fin de mantener la memoria de vídeo libre y disponible para las texturas de los modelos que sí se van a pintar.

void set_reflection_texture(texture_descriptor_t reflectionTex)

Esta función permite al usuario especificar la textura de reflexión que se desea utilizar en cada momento, siendo ésta empleada por todos los modelos que incorporan esta característica. Es imprescindible definir éste parámetro antes de pintar cualquier modelo con reflexión, de lo contrario podría dar lugar a errores.

void bind_reflection_texture()

Esta función genera las textura de reflexión en el hardware mediante la llamada a *glGenTextures*(), cargándola en memoria de video y posibilitando su activación del pintado de una malla con reflexión. Una vez generada la textura, se le asignan los parámetros que le correspondan, de acuerdo al formato de textura y sus dimensiones.

Se le asigna el atributo de generación de coordenadas de textura TEXGEN_NORMAL, con el objetivo de que en el momento del pintado del modelo, las coordenadas de textura se generen teniendo en cuenta las normales del modelo, en vez extraerlas del propio modelo. De esta forma, se consigue ese efecto de deformación según la variación de la superficie del modelo característico de la reflexión.

Estas propiedades se van acumulando en un parámetro entero, para aplicarlos en una sola llamada a *glTexParameter*().

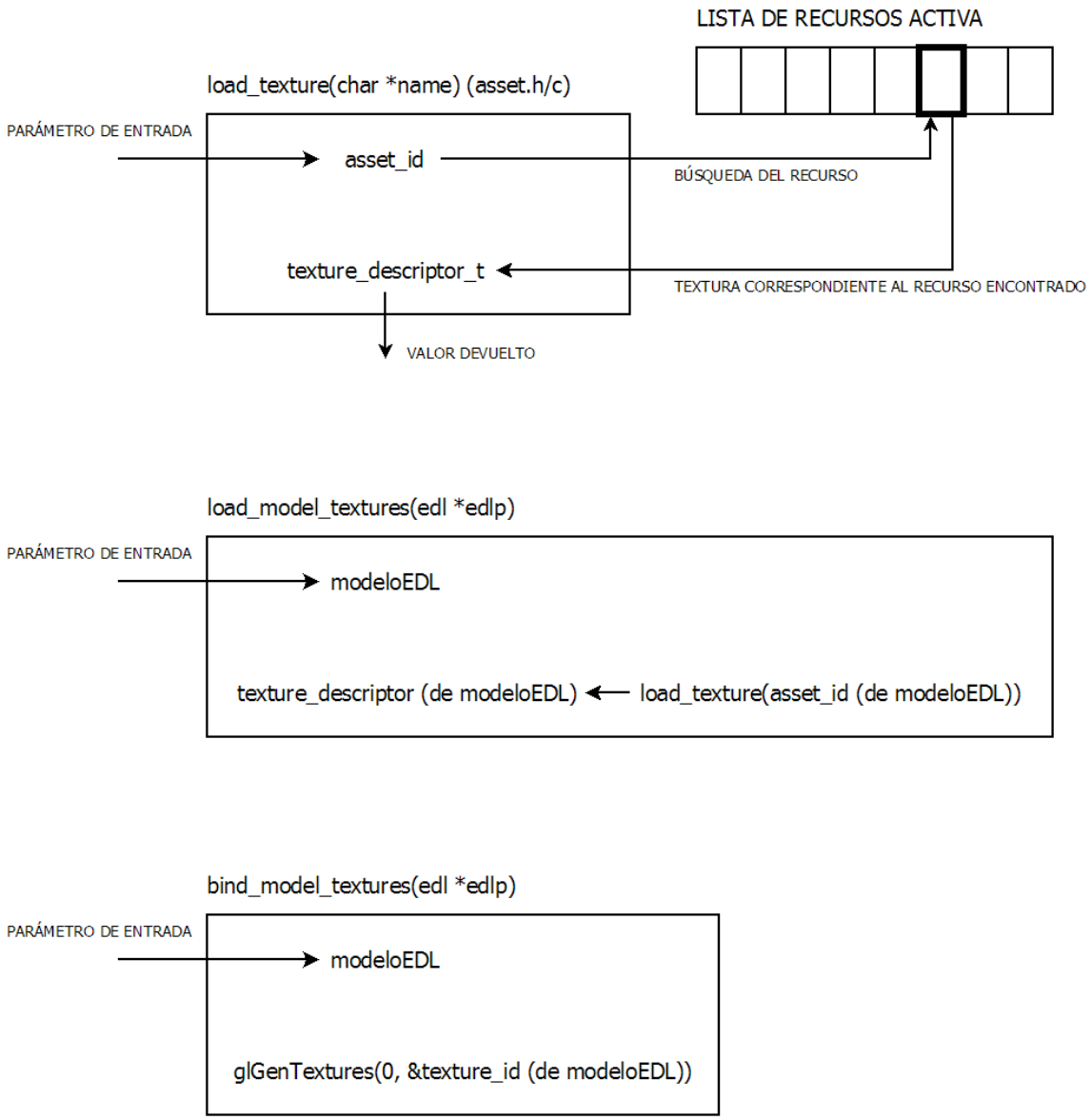

Figura 6.2: Flujo de datos en las funciones de carga y activación de texturas.

Esta función ha de llamarse cada vez que se asigna la textura de reflexión, con *set_reflection_texture*(), y después de haber realizado previamente esta operación, que carga la textura de reflexión en memoria principal.

void unbind_reflection_texture()

Esta función elimina la textura de reflexión de la memoria de vídeo, liberando esta memoria para la carga de otras texturas. Es importante realizar esta operación cada vez que se va asigna una nueva textura de reflexión, con el fin de mantener la memoria de vídeo libre.

void set_anim_speed(edl_instance * edli, int skeleton, float speed)

Esta función de acceso permite al usuario modificar el coeficiente de velocidad de animación de un esqueleto, definido por el parámetro *skeleton*, perteneciente a una instancia de un modelo edl, variando así la velocidad con la que se suceden las muestras de animación para este esqueleto en concreto.

float get_anim_speed(edl_instance * edli, int skeleton)

Esta función de acceso permite al usuario extraer el coeficiente de velocidad de animación de un esqueleto, definido por el parámetro *skeleton*, perteneciente a una instancia de un modelo edl. Puede ser útil para modificar de forma gradual este parámetro, estableciéndolo en la función *set_anim_speed*() como (*get_anim_speed*() + δ), donde δ es la variación de velocidad.

void set_skeleton_anim(edl_instance * edli, int skeleton, int animation)

Esta función de acceso permite al usuario establecer la costumbre de animación de un esqueleto de una instancia de modelo edl. El esqueleto se especifica con el parámetro *skeleton*, mientras que la animación se especifica mediante su índice, definido con el parámetro *animation*.

void set_anim_play_mode(edl_instance * edli, int skeleton, animation_mode animMode)

Esta función de acceso permite al usuario establecer el modo de animación de un esqueleto de una instancia de modelo edl. El esqueleto se especifica con el parámetro *skeleton*, mientras que el modo de animación se especifica mediante *animMode*, que debe ser un elemento del enumerador *animation_mode*.

void set_skeleton_anim_cyclic(edl_instance * edli, int skeleton, int enabled)

Esta función de acceso permite al usuario establecer el estado de animación cíclica de un esqueleto de una instancia de modelo edl. El esqueleto se especifica con el parámetro , mientras que el estado de animación cíclica con *enabled*. Si éste parámetro es 1, se activa la animación cíclica para el esqueleto, si es cero se desactiva.

En el caso en el que se estuviera animando una animación de dicho esqueleto un número determinado de veces, éste proceso se detiene y se continúa con la animación cíclica. Si se quiere activar una nueva animación, previamente hay que llamar a *set_skeleton_anim*() con la costumbre de animación requerida y hacerla cíclica con esta función.

void edli_play_animation_once(edl_instance * edli, int skeleton, int animation, int times, animation_mode animMode)

Esta función de acceso permite al usuario lanzar una animación de un esqueleto de una instancia de modelo edl un número determinado de veces y con un modo de animación

específico. El esqueleto se especifica con el parámetro *skeleton*, la costumbre de animación se especifica mediante su índice, definido con el parámetro *animation*, el número de repeticiones se define con times y el modo de animación con el parámetro *animMode*.

De esta forma, la animación se repetirá el número de veces y bajo el modo de animación especificados, por lo que cuando se hayan reproducido estas repeticiones el modelo quedará inanimado.

int edli_is_animation_repetition_playing(edl_instance * edli, int skeleton)

Esta función de acceso devuelve el estado de repetición de la animación actual de un esqueleto (*skeleton*) de una instancia de modelo edl. De esta forma, si se está reproduciendo una animación un número determinado de veces para este esqueleto, devuelve 1, de lo contrario devuelve 0.

Esta función resulta útil para concatenar animaciones y mantener una continuidad entre éstas, aportando control sobre lo que se está animando.

int edli_is_animation_playing(edl_instance * edli, int skeleton, int animation)

Esta función de acceso devuelve el estado de reproducción de una animación concreta (*animation*) de un esqueleto (*skeleton*), correspondientes a una instancia de modelo edl. De esta forma, si se está reproduciendo la animación especificada devuelve 1, de lo contrario devuelve 0.

Esta función resulta útil para aportar control sobre el estado de animación.

void set_tex_anim_speed(edl_instance * edli, int anim_material, float speed)

Esta función de acceso permite al usuario modificar el coeficiente de velocidad de animación de textura de un material con textura animada correspondiente a una instancia de un modelo edl, variando así la velocidad con la que se suceden las subimágenes de la animación de textura de este material en concreto.

float get_tex_anim_speed(edl_instance * edli, int anim_material)

Esta función de acceso permite al usuario extraer el coeficiente de velocidad de animación de textura de un material con textura animada correspondiente a una instancia de un modelo edl. Puede ser útil para modificar de forma gradual este parámetro, estableciéndolo en la función *set_tex_anim_speed*() como (*get_tex_anim_speed*() + δ), donde δ es la variación de velocidad.

int get_tex_anim_index(edl_instance * edli, char *texture)

Esta función de acceso permite al usuario extraer el índice de una textura animada correspondiente al orden según el cual se procesan las texturas animadas de una instancia de un modelo edl, especificando el nombre de dicha textura (*texture*). Es útil para extraer

y modificar los parámetros de animación de una textura animada (como su velocidad), conociendo únicamente su nombre.

int has_edl_propertie(edl_instance * edli, model_properties propertie)

Esta función de acceso devuelve información de un modelo edl en relación a las propiedades de éste. Así, se le pasa una propiedad de las que puede disponer cualquier modelo edl y la función devuelve 1 en caso de que el modelo especificado contenga dicho atributo, o 0 en caso contrario.

Las propiedades se solicitan mediante los elementos del enumerador *model_properties*, que contiene todos los atributos aplicables a un modelo edl.

edli_translation get_current_joint_translation(edl_instance *edli, int joint)

Esta función de acceso devuelve un parámetro de tipo *edli_translation* con el dato de traslación correspondiente a la muestra actual de animación del esqueleto al que pertenece la articulación de la que se pide el dato (definida por *joint*), de la instancia de modelo edl determinada.

Esta función es útil para extraer la traslación exportada de una articulación, con el fin de posicionar un objeto en la posición de la articulación en cada frame de animación, por lo que dicho objeto seguirá el movimiento de ésta a través de la animación. Así, por ejemplo, se puede posicionar en la mano de un personaje una espada, siendo ésta un modelo independiente.

edli_rotation get_current_joint_rotation(edl_instance *edli, int joint)

Esta función de acceso devuelve un parámetro de tipo *edli_rotation* con el dato de rotación correspondiente a la muestra actual de animación del esqueleto al que pertenece la articulación de la que se pide el dato (definida por *joint*), de la instancia de modelo edl determinada.

Esta función es útil para extraer la rotación exportada de una articulación, con el fin de asignar a un objeto externo al modelo en cuestión la rotación de la articulación en cada frame de animación.

6.2.3. Crear y cargar modelos e instancias

Para pintar un modelo en formato edl cuyos datos han sido previamente cargados en memoria, bien habiéndolos incrustado en el propio programa, o bien en un fichero aparte, lo primero que hay que hacer es crear un modelo edl y cargarlo con la información contenida en el fichero EDL correspondiente al modelo en cuestión. Además, hay que crear al menos una instancia de éste, ya que no se usa el propio modelo en el pintado, sino la instancia, de forma que este proceso sea independiente para cada elemento del modelo en escena.

Para crear un modelo EDL, lo primero es definir una variable de tipo puntero a edl, dónde se almacenarán sus datos. Para cargar los datos e iniciar los parámetros propios

del modelo según sus propiedades se llama a la función *buil_edl*(), pasándole la dirección de los datos del modelo cargado en memoria principal y asignando el puntero a edl devuelto a la variable definida anteriormente. Esta sentencia en el código sería:

```
edl *modelo = build_edl(datos_modelo)
```

siendo *datos_modelo* un puntero que apunta a los datos del modelo cargado en memoria.

De la misma forma, para crear una instancia de dicho modelo se define una variable de tipo puntero a *edl_instance*, a la que se le pasa la dirección de la instancia creada mediante *get_edl_instance*(). De esta forma se asocia el modelo edl a la instancia y se inicializan sus parámetros de animación, en función de las características de éste. A esta función lo único que hay que pasarle es la dirección del modelo edl previamente creado y cargado. Esto es:

```
edl_instance *instancia_modelo = get_edl_instance(modelo)
```

donde *modelo* es el la variable edl previamente definida.

Una vez realizado este proceso de inicialización del modelo edl y su instancia, ya se puede emplear dicho elemento para ser pintado, o como parámetro de entrada a las funciones de acceso disponibles.

6.2.4. Preparar las texturas de un modelo

Con el fin de poder activar las texturas del modelo, operación necesaria en el proceso de renderizado, hay que realizar una serie de operaciones sobre las texturas y las coordenadas de textura de modelo.

Este proceso requiere que previamente se haya iniciado la lista de recursos de tipo textura que se va a emplear y se hayan introducido en ella las texturas en forma de recursos (ver epígrafe anterior).

Cada vez que se cree un modelo y su instancia, para poder pintarlo y que el proceso de texturizado sea correcto, es necesario incluir en el código estas sentencias:

```
load_model_textures(modelo)
convert_texture_coordinates(modelo, 1)
bind_model_textures(modelo)
```

Lo primero que hay que hacer es extraer las texturas asociadas al modelo de la lista de recursos donde se hayan introducido previamente. Este proceso asocia al modelo las texturas cargadas en memoria principal, de forma que cuando se quiera acceder a ellas en el proceso de pintado, se pueda hacer a través del propio modelo. Esta operación se realiza mediante la llamada a la función *load_model_textures*(), pasándole el modelo edl previamente creado y cargado.

Lo siguiente es convertir las coordenadas de textura del modelo recién creado según las dimensiones de la propia textura. Esta operación se realiza mediante la función *convert_texture_coordinates*(), pasándole como parámetros el modelo edl y el modo de conversión, que en este caso es directo, correspondiente al valor 1.

Finalmente, se cargan las texturas del modelo en memoria de video para poder activarlas en el proceso de pintado. Esta operación la efectúa *bind_model_textures*(), recibiendo como parámatro el modelo edl.

Una vez realizado este proceso, las texturas del modelo considerado pueden ser activadas y aplicadas correctamente en el proceso de pintado. Por lo tanto, esta operación hay que realizarla para cada modelo creado que se vaya a pintar (no hay que realizarlo para cada instancia del modelo).

6.2.5. Preparar textura de reflexión

Es necesario inicializar la textura de reflexión en el caso de que algún modelo que se vaya a pintar tenga esta propiedad. Para ello hay que emplear la función *set_reflection_texture*(), que configura la textura global que el sistema de renderizado utilizará para aplicar la capa de reflexión. Esta función se emplea bajo la sentencia:

```
set_reflection_texture(textura_reflexión)
```

donde *textura_reflexión* es un puntero a una variable de tipo *texture_descriptor_t* donde se han cargado previamente los datos de la textura de reflexión.

En el caso en el que se desee cambiar de textura, por cambio de escenario, basta con volver a llamar a la función pasándole los datos de la nueva textura.

6.2.6. Pintar un modelo edl

El pintado de un modelo es muy sencillo con la librería de renderizado msNDS. Es necesario haber creado y cargado un modelo edl y haber instanciado un elemento de éste, además de haber configurado y cargado en memoria de video sus texturas. Además, es conveniente definir una variable entera donde albergar las opciones de renderizado que se vayan a aplicar, asignándole el valor *ro_use_everything* para activar todas las opciones re renderizado disponibles. De esta forma, todas aquellas opciones de las que disponga el modelo (animación, texturas, color por vértice, etc) serán habilitadas y podrán ser configuradas en tiempo de ejecución.

Por lo tanto, para realizar el pintado del modelo en cuestión, basta con incluir en el bucle principal de la aplicación (que se ejecuta cada ciclo a razón de 60 fps en la Nintendo DS) la sentencia siguiente:

```
render_edl(instancia_modelo, opciones_render, atributos_iniciales)
```

A la función que ejecuta el proceso de renderizado *render_edl*() se le pasa la instancia al modelo edl que se quiera pintar (por lo tanto si se instancian varios elementos de un mismo modelo hay que llamar a esta función una vez por cada instancia), la variable entera donde se almacenen las opciones de renderizado y un valor entero que representa los atributos iniciales de polígono para el modelo en general. Debido a la naturaleza general de este último parámetro, en éste simplemente se han de activar los atributos de iluminación de los polígonos del modelo, realizando una suma binaria con los parámetros POLY_FORMAT_LIGHT correspondientes a las luces que se quieran aplicar sobre el modelo. Si se le asignan más atributos de polígono, éstos se aplicarían por igual a todas las mallas del modelo, interfiriendo en aquellos que se aplican en el proceso de renderizado, por lo que es una práctica poco recomendable.

De esta forma, con una sola llamada a esta función se ejecuta todo el proceso de pintado, que puede ser totalmente configurado mediante las opciones de renderizado que

se apliquen en cada momento. Por ello, el usuario no requiere de un nivel elevado de conocimientos de gráficos ni de la Nintendo DS para realizar un renderizado complejo y configurable de la escena.

6.2.7. Control de animación

Controlar la animación de la instancia de un modelo edl que se está renderizando requiere varias operaciones, que dependen de la aplicación y el tipo de animación que se requiera. El sistema de animación permite un amplio control sobre los aspectos clave de ésta, como definir una animación cíclica o reproducirla un número determinado de veces, cambiar la costumbre de animación de un esqueleto aislado del modelo, cambiar su modo de animación o su velocidad de reproducción.

Iniciar animación

Un modelo edl que contenga datos de animación se inicia por defecto inanimado, de modo que es el usuario el que se tiene que encargar de activar la animación, bien al inicio de la ejecución, o bien en un momento específico determinado por el propio usuario de la librería o por el jugador.

Para ello hay dos formas. Si se requiere que el esqueleto de un modelo inicie su animación y se anime indefinidamente, hay que, por un lado definir la costumbre de animación que se quiere reproducir, y por otro activar la animación cíclica. Esto se realiza con las sentencias:

```
set_skeleton_anim(instancia_modelo, esqueleto, animación)
set_skeleton_anim_cyclic(instancia_modelo, esqueleto, 1)
```

siendo *esqueleto* el índice al esqueleto del cual se quiere activar la animación y *animación* la costumbre de animación que se desea reproducir. Se pasa 1 como parámetro de activación a la función *set_skeleton_anim_cyclic*() para dejar el esqueleto en estado de animación cíclica.

De esta forma, el modelo se animará de forma permanente, hasta que se pare la animación cíclica de dicho esqueleto llamando a la función *set_skeleton_anim_cyclic*() pasándole como parámetro de activación el valor 0. En este instante la animación se detendrá, sin esperar al final del cíclo.

En el caso de que se quiera activar una animación un número finito de veces, ha de emplearse la función *edl_play_animation_once*(), cuya sentencia sería:

```
edl_play_animation_once(instancia_modelo, esqueleto, animación, repeticiones, modo)
```

siendo *repeticiones* el número de veces que ha de repetirse la animación y *modo* el modo de animación (hacia delante, hacia detrás o *ping-pong*). De esta forma, cuando la animación se haya reproducido el número de veces especificado, ésta se detendrá, quedándose el modelo estático en la posición de reposo hasta que se vuelva a activar una animación.

Hay que realizar este proceso para cada esqueleto del modelo que se quiera animar de forma independiente, por lo que pueden animarse algunos esqueletos y otros no.

Cambiar costumbre de animación de un esqueleto

Para cambiar la costumbre de animación de un esqueleto de un modelo edl basta con llamar a la función *set_skeleton_anim*() pasándole el índice del esqueleto del cual se quiere cambiar la costumbre y el índice de dicha costumbre.

Si la instancia del modelo edl que se estaba animando estaba haciéndolo cíclicamente y no se cambia este estado previamente, la animación comenzará inmediatamente y se reproducirá indefinidamente. Si no estaba en este estado y se requiere que lo esté, ha de llamarse previamente a la función *set_skeleton_anim_cyclic*(), pasándole 1 como parámetro de activación.

También se puede cambiar la costumbre de animación de un esqueleto y que ésta se repita un número determinado de veces llamando a la función *edl_play_animation_once*(), pasándole el índice al esqueleto, el índice de la animación que se quiera activar, el número de repeticiones y el modo de reproducción de la animación.

Cambiar modo de animación de un esqueleto

Para cambiar el modo de reproducción de la animación de un esqueleto hay que llamar a la función *set_anim_play_mode*(), cuya sentencia sería:

```
set_anim_play_mode(instancia_modelo, esqueleto, modo)
```

De esta forma, el modo de animación del esqueleto especificado cambia instantáneamente al definido con el parámetro *modo*, continuando la animación en este estado.

Cuando se llama a la función *edl_play_animation_once*() se determina el modo de animación que se va a emplear en las repeticiones, quedándose este estado definido para siguientes reproducciones cíclicas. Por lo tanto, en el caso en el que se desee cambiar el modo de animación para una animación reproducida indefinidamente, hay que activar dicho modo antes de activar el estado cíclico de la animación.

Cambiar velocidad de animación de un esqueleto

El sistema de animación permite ajustar la velocidad a la que se reproduce la animación de un esqueleto perteneciente una instancia de modelo edl. De esta forma, los frames de la animación se sucederán a un periodo distinto al correspondiente al modelo por defecto, según la velocidad establecida sea mayor o menor.

Para ello se ha puesto a disposición la función *set_anim_speed*(), aplicándose mediante la sentencia:

```
set_anim_speed(instancia_modelo, esqueleto, velocidad)
```

donde el parámetro *velocidad* es un número flotante que indica el coeficiente de variación de velocidad. Así, el valor 1 es la velocidad por defecto, correspondiendo al 100 %, y valores por encima y por debajo varían este porcentaje.

Sin embargo, el sistema de animación trabaja de la siguiente forma: extrae el periodo de animación almacenado en el modelo edl, que representa el periodo en frames de la Nintendo DS (recordemos que es 60 fps), es decir el número de ciclos que tienen que transcurrir para actualizar a la muestra de animación siguiente, y se divide por el coefi-

ciente de velocidad (cuanto mayor sea este número, menor será el periodo y más rápido se actualizará la animación); tras esto, se incrementa un contador de frames; si este contador es mayor que el periodo (si han transcurrido ese número de ciclos) entonces se actualiza la muestra de animación (que dependiendo del modo de animación, será la siguiente o la anterior), sino, se sigue incrementando dicho contador.

Esto implica que variaciones pequeñas de este coeficiente pueden no traducirse en un aumento de la velocidad de animación, ya que se tiene en cuenta la parte entera del periodo, por lo que si se reduce unas décimas el resultado es el mismo. Al aumentar la velocidad (reducir el periodo) este efecto es más notorio, sobre todo si el periodo es de por sí pequeño, ya que pequeñas variaciones fluctuarán entre dos mismos números enteros. Sin embargo, en el caso de reducir la velocidad (aumentando para ello el periodo) pequeñas variaciones del coeficiente se traducen en rápidos cambios de velocidad, ya que un número pequeño dividido por un uno inferior a 1 aumenta rápidamente.

Para realizar variaciones sobre la velocidad actual, se puede obtener este dato llamando a la función *get_anim_speed*(), bajo la sentencia:

```
get_anim_speed(instancia_modelo, esqueleto)
```

De esta forma se puede variar la velocidad en base a la que lleve en cada momento.

Cambiar velocidad de animación de textura de un material animado

Al igual que la velocidad de animación de un esqueleto, se puede variar la velocidad de animación de una textura animada por subimágenes. Para ello se emplea la función *set_tex_anim_speed*(), a la que se le pasa el índice al material que contiene animación de textura al que se desea acceder dentro del array de texturas animadas de la instancia del modelo. Dado que este parámetro se autogenera dependiendo del número de materiales animados del modelo, se puede acceder a él mediante la función *get_tex_anim_index*(), que devuelve dicho parámetro pasándole el nombre del material con textura animada.

De esta forma, las sentencias del código serían

```
int material_animado = get_tex_anim_index(instancia_modelo, nombre)
set_tex_anim_speed(instancia_modelo, material_animado, velocidad)
```

donde el parámetro *nombre* es el nombre del material del cual se quiere extraer el índice y *material_animado* es el índice al material que contiene animación de textura devuelto por *get_tex_anim_index*().

Para realizar variaciones sobre la velocidad actual, se puede obtener este dato llamando a la función *get_tex_anim_speed*(), bajo la sentencia:

```
get_tex_anim_speed(instancia_modelo, material_animado)
```

6.2.8. Extraer transformaciones de articulación exportada

Con el fin de aplicar a un modelo la transformación de una articulación correspondiente a un modelo diferente, en cada muestra de animación de éste último, es necesario poder almacenar dichos datos en variables, para a continuación aplicar la traslación y la rotación correspondientes. Para esto se han dispuesto las funciones *get_current_joint_translation*() y *get_current_joint_rotation*() respectivamente.

Para almacenar éstos parámetros en cada ciclo de la aplicación, basta con definir dos variables de tipo puntero a *edl_translation* y *edl_rotation* y almacenar en ellas el valor devuelto por dichas funciones. El proceso es el siguiente:

edl_translation *traslación = get_current_joint_translation(instancia_modelo_ref, articulación)

edl_ rotation *rotación = get_current_joint_rotation(instancia_modelo_ref, articulación)

donde *instancia_modelo_ref* es el elemento del modelo edl del cual se quiere extraer la transformación, y *articulación* es la articulación de este modelo de la cual se quiere conocer esta información.

Si esto se ejecuta en cada ciclo y el modelo de referencia está animado, el objeto al que se le aplique esta transformación se posicionará en el lugar de la articulación especificada y rotará según lo haga ésta, en cada muestra de la animación.

Hay que tener presente que la articulación de la cual se quiere extraer la transformación debe tener éstos datos disponibles, bien porque esté animada, o bien porque se haya especificado expresamente la importación de su traslación y su rotación en el momento del modelado en MilkShape 3D, mediante las directivas #EP y #ER respectivamente, las cuales son interpretadas por conversor EDL, que añade al modelo esta información.

6.3. Edl.h: funcionamiento interno

Dado que el sistema de renderizado es considerablemente complejo, varios sub-procesos de éste se han independizado para hacerlo más modulable y que el código quede más limpio y accesible a la hora de desarrollar y depurar el programa.

Estos sub-procesos corresponden a las funciones internas que se emplean en el proceso de renderizado. Éstas no deben ser utilizadas por el usuario de la librería, ya que no están habilitadas a tal efecto y podría ser causa de fallos o simplemente no funcionarían correctamente.

A continuación se exponen los tipos de datos y las funciones que componen este sistema interno de la librería msNDS, además de unas definiciones añadidas y unas variables globales (correspondientes a la textura de reflexión empleada por el sistema).

6.3.1. Definiciones

En primer lugar, se han definido dos identificadores que complementan a los elementos del enumerador GL_POLY_FORMAT_ENUM definido en el módulo videoGL de Libnds, ya que éstos han sido obviados por razones desconocidas. Como ya se ha explicado, éstos acceden a los sectores del comando de atributos del polígono, correspondiente al hardware de la Nintendo DS. Éstos son:

- **POLY_SET_NEW_DEPTH_IF_TRANS = 1<<11**: establece que los píxels translúcidos se tengan en cuenta en el buffer de profundidad, anteponiéndose a píxels con profundidad menor. Normalmente este parámetro no se usa, ya que no es deseable que los polígonos translúcidos alteren el buffer de profundidad, causando que los objetos opacos detrás de objetos translúcidos no se pinten.

- **POLY_DEPTH_TEST_EQUAL = 1<<14**: habilita el test de profundidad a cierto para profundidad igual, no menor, que es el valor por defecto. Esto significa que solo se pintan los píxels cuya profundidad sea igual a los que haya en el buffer de profundidad. Este parámetro es esencial para realizar el pintado multi-pasada o multi-capa, en el que se envían varias veces los vértices del modelo, con propiedades diferentes en cada pasada. Estableciendo este parámetro se consigue que los vértices reenviados se vuelvan a pintar (en caso contrario se obvian, ya que el test de profundidad da negativo al tener la misma profundidad, y no menor).

6.3.2. Variables globales

Se han definido dos variables globales en las que se almacenan los datos de la textura de reflexión que se desee emplear y su identificador generado con *glGenTextures*(). Dado el sistema de reflexión simple que se ha implementado (no se realiza la reflexión teniendo en cuenta los demás objetos de la escena, sino que se aplica una textura que representa el entorno donde está situado el objeto reflectante) tiene sentido disponer de una sola textura de reflexión para todos los objetos reflectantes de la escena, ya que todos están en el mismo entorno.

Por lo tanto, al aplicar la capa de reflexión, si la malla pintada tiene la propiedad reflectante, se activa esta textura, sumándose a la que pudiera tener previamente la malla.

Estas variables globales son:

- **texture_descriptor_t * REFLECTION_TEXTURE**: estructura de tipo textura que almacena los datos de la textura de reflexión.
- **int *REFLECTION_TEXTURE_ID**: almacena el identificador generado con *glGenTextures*() para la activación de la textura de reflexión.

6.3.3. Estructuras y tipos de datos internos

Con el fin de almacenar los datos contenidos en el fichero EDL (cuya estructura se ha detallado en el capítulo anterior) se han definido una serie de estructuras que corresponden a las diferentes secciones del modelo EDL. Así, la estructura del modelo contiene diferentes parámetros cuyos tipos corresponden a estas estructuras, de forma que tanto la estructura como el acceso sean modulares.

Así, un modelo EDL tiene diferentes parámetros correspondientes a los esqueletos que lo componen, sus articulaciones, las mallas que construyen su geometría y sus materiales asociados. Todos estos elementos están implementados, como se detalla a continuación, por estructuras que contienen sus datos y características. En la Figura 6.1 se muestra la estructuración de estos datos, para tener en todo momento claro esta organización de éstos.

edl_mesh

- **int nvert**: número de vértices que componen la malla.
- **int nnorms**: número de normales que componen la malla.

- **int ntris**: número de triángulos que componen los vértices de la malla.
- **int material_index**: índice del material asociado a la malla.
- **int flags**: banderas que determinan ciertas propiedades de la malla. (Ver especificación EDL).
- **int skeleton**: índice al esqueleto que actúa sobre la malla. En caso de no tener ningún esqueleto asociado su valor será 255.
- **int joint**: índice a la articulación que controla la malla. Si su valor es 255, indica que la malla no puede animarse por transformación. Si el valor de *skeleton* es 255 (ningún esqueleto asociado), este valor es ignorado.
- **unsigned int * vertices**: puntero que apunta a la dirección de memoria donde comienza la lista de coordenadas de vértices en el fichero del modelo EDL cargado en memoria.
- **unsigned int * normals**: puntero que apunta a la dirección de memoria donde comienza la lista de normales en el fichero del modelo EDL cargado en memoria.
- **unsigned int * texcoords**: puntero que apunta a la dirección de memoria donde comienza la lista de coordenadas de textura en el fichero del modelo EDL cargado en memoria.
- **unsigned int * tri_verts**: puntero que apunta a la dirección de memoria donde comienza la lista de triángulos con los índices a los vértices que los componen en el fichero del modelo EDL cargado en memoria.
- **unsigned int * tri_norms**: puntero que apunta a la dirección de memoria donde comienza la lista de triángulos con los índices a las normales que los componen en el fichero del modelo EDL cargado en memoria.
- **unsigned int * cpv_colors**: puntero que apunta a la dirección de memoria donde comienza la lista de colores por vértice en el fichero del modelo EDL cargado en memoria.
- **unsigned int * vertex_samples**: puntero que apunta al comienzo de la lista de muestras de todos los vértices animados por muestreo (no por transformación). El número de muestras será $Nsamples = nvert \times nsamples$ (del esqueleto).
- **unsigned int * normal_samples**: puntero que apunta al comienzo de la lista de muestras de todas las normales animadas por muestreo (no por transformación). El número de muestras será $Nsamples = nnorms \times nsamples$ (del esqueleto).

Esta estructura contiene los datos y las propiedades de una malla del formato EDL. Por lo tanto, cada una de las mallas del modelo viene representada por un parámetro de este tipo, con sus propiedades particulares.

El modelo EDL es cargado completamente en memoria o de un fichero. De esta forma, los parámetros *vertices*, *normals*, *txcoords*, *tri_verts*, *tri_norms*, *cpv_colors*, *vertex_samples* y *normal_samples* son punteros a enteros que apuntan al inicio de la lista de sus respectivos datos. De esta forma, se accede directamente a los datos del modelo cargado en memoria en vez de realmacenar toda esta información de nuevo en arrays, con el consumo de memoria que esto supone.

En esta estructura hay datos que, de no estar la maya animada, son inexistentes, tales como las listas de muestras de vértices y normales (este último además es opcional), o el

índice a la articulación, en caso de que no tenga esqueleto asociado. Además, si la maya no tiene asociadas coordenadas de textura o colores por vértice, sus datos respectivos también son ignorados. En el caso en el que la maya no contenga alguno de estos datos opcionales, sus campos respectivos se ponen a NULL, con el fin de controlar su presencia en el momento del pintado.

El parámetro *flags* contiene, en un solo entero, información relativa a si la maya es sólida (para hacer o no *backface culling*), si incluye normales, si incluye coordenadas de textura, si incluye color por vértice, el formato de los vértices que componen la malla (V10 o V16), si las coordenadas de textura en horizontal y en vertical necesitan repetición y si la malla está animada. Estos estados se controlan aplicando a este parámetro una máscara de bit cuyo valor se extrae del enumerador *mesh_flags* (se describe a continuación), conteniendo todas estas posibilidades.

Estas banderas son imprescindibles para conocer en la etapa de pintado qué propiedades tiene la malla y operar en consecuencia. De esta forma, por ejemplo, si el bit 2 del parámetro *flags* está a cero (se contrasta aplicando el parámetro *mf_has_texcoords*), quiere decir que en la etapa de lectura del modelo estos datos han de obviarse, ya que no están presentes en el archivo. Además en la etapa de pintado el proceso de texturizado también se obviará. Es una herramienta muy versátil, ya que el mismo modelo almacena información de renderizado.

mesh_flags

- **mf_solid = 1**
- **mf_has_normals = 2**
- **mf_has_texcoords = 4**
- **mf_cpv = 8**
- **mf_vertex_v16 = 16**
- **mf_h_texcoord_repeat = 32**
- **mf_v_texcoord_repeat = 64**
- **mf_animated = 128**

Como se ha adelantado, las banderas son parámetros útiles para conocer información del modelo y procesarlo según sea necesario. En este caso, *mesh_flags* es un enumerador que contiene los parámetros necesarios para realizar las máscaras de bit sobre el parámetro *flags* de la estructura *edl_mesh*.

De esta forma, con *mf_solid* se accede al bit 0 del parámetro, con *mf_has_normals* se accede al bit 1 (2 en binario es 10) y así sucesivamente. Realizando una máscara de bits, se comprueba si dicho bit está a 0 o a 1, comprobando así si el dato está activo o no.

edl_material

- **unsigned int ambient_difusse**: componentes ambiente y difuso del material (15 bits cada uno, 5 bits cada componente rgb), tal y como se manda el comando al hardware.

- **unsigned int especular_emissive**: componentes especular y emisivo del material (16 bits cada uno, 5 bits cada componente rgb), tal y como se manda el comando al hardware.
- **int transparency**: transparencia del material de 0 (diáfano) a 31 (opaco).
- **int shininess**: amplitud del resalte especular del material. En la última versión de EDL, se ha eliminado este parámetro, manteniéndose en la estructura por compatibilidad con versiones anteriores.
- **int reflection_alpha_blend**: transparencia de mezcla de la capa de reflexión (0 a 31).
- **int scnd_tex_alpha_blend**: transparencia de mezcla de la capa de segunda textura (0 a 31).
- **int anim_period**: periodo de animación en frames NDS de la animación de la primera textura (si ésta está animada).
- **int si_X**: número de subimágenes en horizontal de la textura animada.
- **int si_Y**: número de subimágenes en horizontal de la textura animada.
- **int flags**: banderas que determinan ciertas propiedades del material. (Ver especificación EDL).
- **char asset_id[13]**: identificador de la textura principal para buscarla en el gestor de recursos.
- **char asset_scnd_tex_id[13]**: identificador de la segunda textura para buscarla en el gestor de recursos.
- **texture_descriptor_t * texture_descriptor**: puntero a la textura principal.
- **texture_descriptor_t * scnd_texture_descriptor**: puntero a la segunda textura.
- **int texture_id**: identificador de la textura principal generado con *glGenTextures*() para activar la textura.
- **int scnd_texture_id**: identificador de la segunda textura generado con *glGenTextures*() para activar la textura.

Esta estructura contiene los datos y las propiedades de un material del formato EDL. Por lo tanto, cada uno de los materiales del modelo viene representado por un parámetro de este tipo, con sus propiedades particulares.

Básicamente contiene los datos albergados en el archivo EDL en relación a las mallas, comentados en el capítulo anterior, con los tipos de datos necesarios en cada caso.

Las texturas no se almacenan en la estructura, sino que se almacena su dirección de memoria, pudiendo acceder a ellas en cada momento, pero ahorrando este espacio.

El parámetro *flags* contiene las banderas que aportan información del material en relación a si éste tiene identificador de textura (es decir, si tiene textura asociada), si tiene segunda textura, si la primera textura está animada y si esta animación debe ser cíclica. Estos estados se controlan aplicando a este parámetro una máscara de bit cuyo valor se extrae del enumerador *material_flags*, conteniendo todas estas posibilidades.

material_flags

- **mf_has_texture = 1**
- **mf_has_scnd_texture = 2**
- **mf_has_first_texture_animated = 4**
- **mf_cyclic_animation = 8**

El enumerador *material_flags* contiene los parámetros necesarios para realizar las máscaras de bit sobre el parámetro *flags* de la estructura *edl_material* y así comprobar las características propias del material, de tal forma que se realice un proceso u otro.

edl_skeleton

- **int nsamples**: número de muestras de la animación completa.
- **unsigned short period**: periodo de muestreo de la animación en frames NDS.
- **int nanim**: número de costumbres de animación.
- **unsigned short *final_samples**: puntero al inicio de la lista de frames finales de cada costumbre de animación, en el caso de que tenga más de una. En el caso de que solo tenga una costumbre, la muestra final será (*nsamples* – 1).

Esta estructura contiene los datos y las propiedades de un esqueleto del formato EDL. Por lo tanto, cada uno de los esqueletos del modelo viene representado por un parámetro de este tipo, con sus propiedades particulares.

En esta estructura, básicamente se almacenan los datos de la línea temporal de animación, es decir, el número de muestras que ésta tiene, su periodo (en frames NDS, es decir, cada cuántos ciclos en la consola hay que avanzar al frame siguiente, teniendo en cuenta que ésta va a 60 frames por segundo fijos), el número de costumbres de animación contenidas en ésta y, en caso de tener más de una, las muestras finales de cada una de estas costumbres (con el fin de acotarlas en el tiempo).

Cada uno de los esqueletos del modelo es independiente en términos de animación, como se explicará más en detalle en el apartado de *edl_instance*. Por ello, es posible animar por separado y de diferentes modos cada uno de estos esqueletos, aunque estos pertenezcan al mismo modelo. Esta es una característica interesante, ya que es posible animar, por ejemplo, diferentes objetos de un escenario, como pudieran ser unas plataformas por las que tiene que ir saltando el jugador, o las puertas y ventanas de una habitación, manejando un único modelo, lo que simplifica la tarea de desarrollo.

edl_joint

- **int skeleton_id**: identificador del esqueleto al que pertenece la articulación.
- **int flags**: banderas que determinan ciertas propiedades de la articulación. (Ver especificación EDL).
- **unsigned int *idle_traslation**: puntero a la traslación de la articulación en reposo, en formato f32. Necesaria para cuando el modelo no se esté animando.
- **unsigned int *sample_traslations**: puntero al inicio de la lista de traslaciones

de la articulación en cada muestra de animación, en el caso en el que esté animada.

- **unsigned int *sample_rotations**: puntero al inicio de la lista de rotaciones de la articulación en cada muestra de animación, en el caso en el que esté animada.

Esta estructura contiene los datos y las propiedades de una articulación del formato EDL. Por lo tanto, cada una de las articulaciones del modelo viene representada por un parámetro de este tipo, con sus propiedades particulares.

La estructura *edl_joint* se emplea para almacenar las transformaciones en cada muestra de animación, en el caso en el que disponga de esta funcionalidad. De esta forma, de estar la articulación animada, en cada una de las muestras de la animación se le aplica a las mallas asociadas a esta articulación la transformación dada por la traslación y la rotación en dicha muestra. Este método sólo se emplea en el caso en el que dicha malla esté únicamente controlada por esta articulación, es decir en la animación por transformación.

Los datos de muestreo se extraen accediendo al esqueleto especificado por el parámetro *skeleton_id*, que es el que controla la articulación considerada.

Como en los casos anteriores, el parámetro *flags* contiene las banderas que aportan información de la articulación en relación a si ésta está afectada por un *billboard* esférico o cilíndrico, si tiene información de traslación en cada muestra de animación o si tiene información de rotación en cada muestra de animación. De esta forma, si dispone de transformaciones por muestra, se aplicará el proceso de transformación, en caso contrario éste se obviará, tanto en el momento de lectura del modelo como en el del pintado. Estos estados se controlan aplicando a este parámetro una máscara de bit cuyo valor se extrae del enumerador *joint_flags*, conteniendo todas estas posibilidades.

joint_flags

- **jf_cilindric_BB = 1**
- **jf_spheric_BB = 2**
- **jf_has_translation = 4**
- **jf_has_rotation = 8**

El enumerador *joint_flags* contiene los parámetros necesarios para realizar las máscaras de bit sobre el parámetro *flags* de la estructura *edl_joint* y así comprobar las características propias de la articulación, de tal forma que se realice un proceso u otro.

model_flags

- **mf_has_reflective_materials = 1**
- **mf_has_material_scnd_tex = 2**
- **mf_refletion_2nd_tex_order = 4** (1: 2ª textura/reflexión 0: reflexión/2ª textura)
- **mf_has_model_textures = 8**
- **mf_has_model_animation = 16**
- **mf_has_model_cpv = 32**

- **mf_has_model_materials = 64**
- **mf_has_model_normals = 128**
- **mf_has_model_texture_animation = 256**
- **mf_has_model_billboard = 512**

El enumerador *model_flags* contiene los parámetros necesarios para realizar las máscaras de bit sobre el parámetro *flags* de la estructura del modelo edl (que se detalla en el epígrafe siguiente) y así comprobar las características propias del modelo edl, de tal forma que se realice un proceso u otro.

En este caso, los elementos que van desde *mf_has_model_textures* en adelante no se extraen del fichero EDL, sino que se añaden al parámetro *flags* en el momento de la lectura, según se aplican las características correspondientes, con el fin de aportar esta información adicional y tener un mayor acceso a las propiedades del modelo.

render_pass_type

- **RENDER_TYPE_DEFAULT**
- **RENDER_TYPE_SCND_TEXTURE**
- **RENDER_TYPE_REFLECTION**

Este enumerador interno contiene los diferentes modos de pintado que han de especificarse a la hora de pintar un modelo cuando se llama, en el sistema de render, a la función *render_pass*() (y las funciones llamadas de forma consecutiva a través de ésta), determinando el tipo de pasada que se quiere realizar. De esta forma, si se llama a la función *render_pass*() pasándole como parámetro el tipo de pasada RENDER_TYPE_DEFAULT, se realizará el pintado del modelo en modo normal, es decir aplicando su material, con su transparencia, y la primera textura si tiene. En caso de hacerlo pasándole RENDER_TYPE_SCND_TEXTURE, se hará un pintado de los mismos vértices, con la salvedad de activar la segunda textura en vez de la primera, y obviando la información de material, aplicándole la transparencia de mezcla de segunda textura. Lo mismo ocurre en el caso de realizar una pasada en modo RENDER_TYPE_REFLECTION, dónde se activa la textura de reflexión especificada previamente, se obvia la información material y se aplica la transparencia de mezcla de reflexión, consiguiendo una mezcla con las propiedades aplicadas en las capas anteriores.

6.3.4. Funciones internas

Al igual que para las estructuras y tipos de dato, las funciones definidas en el módulo edl se pueden clasificar de dos formas: las internas, empleadas por el sistema de renderizado, a las que el usuario no debe acceder por motivos de diseño y estabilidad, y las de usuario, es decir las que éste debe emplear para acceder a las capacidades del sistema.

En este epígrafe se enumeran las funciones internas y se describe brevemente su funcionamiento. Para más detalle sobre éste se recomienda acceder al código.

edl * build_edl0(unsigned int * data)

Esta función recibe los datos de un modelo edl, comprobando que el formato es correcto, y los lee, almacenándolos en las estructuras que componen el modelo edl y devolviendo un puntero al modelo edl generado.

El proceso de lectura generalizado (no exclusivo de esta versión) es el siguiente:

- se define una variable entera que corresponde al parámetro considerado en cada momento. Éste se corresponde con la estructura de 32 bits bajo la cual están almacenados los datos en el fichero edl, tal y como se ha explicado en el capítulo referente a dicho formato.
- se define un puntero que apunta en cada momento a la posición del bloque de 32 bits considerado.
- se procesa el parámetro actual, atendiendo a la estructura del fichero edl (en cada momento se sabe qué se está leyendo, ya que se conoce la estructura y orden de almacenamiento de los datos y el número de elementos de articulaciones, esqueletos, mallas, vértices, normales y materiales presentes en el modelo):
  - en el caso en el que el parámetro considerado sea un valor concreto, éste se almacena en el parámetro de la estructura correspondiente
    - si el parámetro contiene varios datos (como es el caso de las opciones de malla, en las que las banderas, el índice al esqueleto y el índice a la articulación están almacenados en un único bloque de 32 bits), se considera cada uno separadamente mediante un desplazamiento y una máscara de bits correspondientes al tamaño de cada dato (el tamaño de cada dato está definido en la especificación del formato del Anexo A).
  - en el caso en el que el parámetro corresponda al inicio de una lista de elementos (vértices, normales, coordenadas de textura, colores por vértice, vértices por muestra, normales por muestra y transformaciones de articulaciones por muestra) se asigna la dirección del primer parámetro de la lista al parámetro en la estructura correspondiente (en el caso de las coordenadas de vértice se asigna al puntero *vertices* de la malla correspondiente) y se incrementa el parámetro el número de elementos de dicha lista (*nvertices* de la malla correspondiente en el ejemplo anterior), saltando este número de elementos, para considerar el dato siguiente.
- una vez procesado el parámetro, se incrementa su valor, pasando a procesar el bloque de 32 bits siguiente.

Al final de este proceso, cada uno de los parámetros de las estructuras que componen el modelo edl quedan asignados con los valores correspondientes a los datos almacenados en el fichero EDL.

En el caso de esta función de lectura, se leen únicamente ficheros de datos EDL cuya versión sea la 0, teniendo en cuenta la cabecera de éste (el primer bloque de 32 bits). Por lo tanto se leen únicamente los datos presentes en esta versión, quedando el resto de datos sin asignar.

edl * build_edl1(unsigned int * data)

Esta es la función de lectura del modelo cuya versión del formato sea la 1, con el procesado de datos correspondiente a su estructura.

edl * build_edl2(unsigned int * data)

Esta es la función de lectura del modelo cuya versión del formato sea la 2, con el procesado de datos correspondiente a su estructura.

edl * build_edl3(unsigned int * data)

Esta es la función de lectura del modelo cuya versión del formato sea la 3, con el procesado de datos correspondiente a su estructura. Siendo la última versión implementada, es la única que tiene en cuenta todos los datos correspondientes al modelo (incluyendo datos de animación).

void render_pass(edl_instance * edlip, int options, u32 poly_attr, render_pass_type render_type)

El sistema de renderizado está subdividido en diferentes procesos con el fin de facilitar la repetición de operaciones empleando diferentes parámetros. De esta forma, el sistema es más modular y el código es más claro y sencillo, tanto de manejar como de modificar, dado que las funciones que intervienen realizan procesos más específicos.

De esta forma, en el sistema se concatenan una serie de funciones que se van llamando según la etapa del proceso y las propiedades de lo que se va a pintar. Así, la función que nos ocupa, *render_pass*(), es la encargada de hacer una pasada de render atendiendo al modo de pintado que se le haya pasado (*render_type*), llamando a su vez a la función *render_mesh*() con las propiedades de la malla considerada, procesando primero las mallas opacas y más tarde las translúcidas. Si se están pintando las pasadas de segunda textura o de reflexión, se analiza la transparencia de mezcla respectiva. Si ésta es nula (porque alguna parte del modelo tenga esta propiedad pero la malla considerada no), entonces no se manda pintar la malla considerada, debido, en primer lugar a que no tiene sentido hacerlo, ya que esta malla no goza de dicha propiedad, y segundo porque si se hace y se le asigna a la malla pintada una transparencia de 0, ésta se pintará en modo alambre, produciendo un efecto indeseado.

A su vez, en la función de render global, *render_edl*() (que se explicará más adelante, ya que es una función de usuario), se hacen las llamadas a *render_pass*() necesarias según el modelo requiera de estas pasadas de pintado, porque incorpore una segunda textura o reflexión (considerando, en el caso en el que incorpore ambas funcionalidad, el orden de pintado de cada pasada). En la Figura 6.3 se muestra esta concatenación de procesos.

A través de las funciones se van traspasando los datos de tipo de pasada de render, atributos del polígono, opciones de renderizado y, a partir del pintado de mallas, el número de la malla considerada, para disponer en cada etapa de la información básica de los polígonos que se van a pintar.

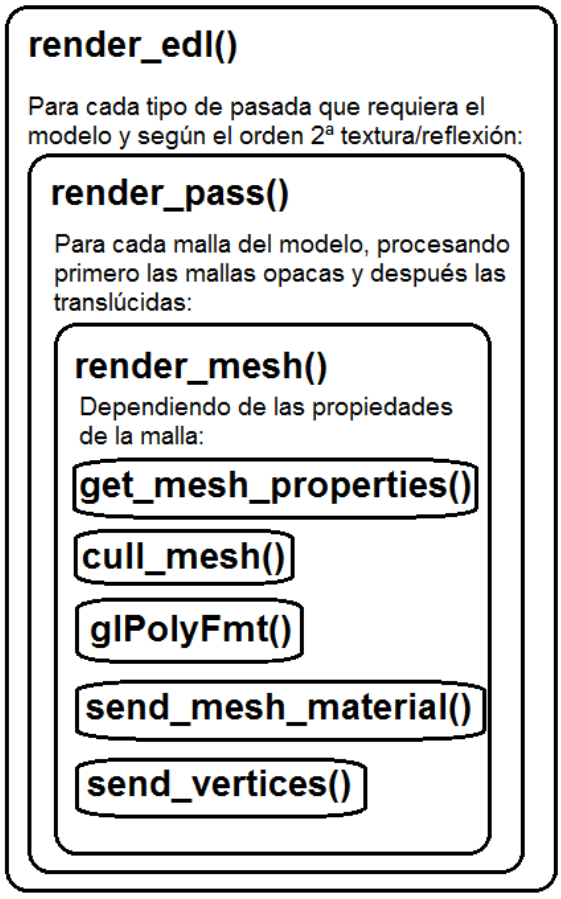

Figura 6.3: Estructuración de las llamadas a las funciones del proceso de renderizado.

void render_mesh(edl_instance * edlip, int meshN, int options, u32 poly_attr, render_pass_type render_type)

En esta función del sistema de renderizado, se procesa la malla considerada, cuyo índice viene definido por *meshN*. Según el tipo de pasada, definido por *render_type*, los atributos del polígono, *poly_attr*, y las opciones de renderizado, *options*, se realizan unas operaciones u otras.

Lo primero que se efectúa, tal y como muestra la Figura 6.3, es la obtención de las propiedades del polígono, asignadas al parámetro *poly_attr*. Éstas se extraen llamando a la función *get_mesh_properties*(), donde se consideran tanto las propiedades específicas del modelo edl, como las opciones de renderizado.

Una vez obtenido este parámetro, se realiza el descarte de las caras traseras, que depende de si el objeto es sólido o no. Esta propiedad, como ya se ha comentado, viene definida en las banderas de la malla. El proceso se realiza sumando al parámetro *poly_attr* POLY_CULL_NONE, en el caso de ser un objeto no sólido (una caja abierta o un plano), o POLY_CULL_BACK, en el caso de serlo.

A continuación se procesa la transparencia, que dependerá de la pasada de render, extrayendo en cada caso la transparencia del material, la transparencia de mezcla de segunda textura o la transparencia de mezcla de reflexión.

En el caso de que se esté realizando una pasada de reflexión o de segunda textura, se le añade a los atributos del polígono el parámetro POLY_DEPTH_TEST_EQUAL, habilitando el pintado de píxels con profundidad igual. De esta forma se permite el pintado de la pasada de render.

Una vez que se han extraído todos los atributos del polígono necesarios, se llama a la función *glPolyFmt*(), aplicándolos estableciendo el estado del hardware.

A continuación se aplica el material correspondiente a la malla, llamando a la función *send_mesh_material*().

Finalmente, se mandan los vértices llamando a la función *send_vertices*(), aplicando la transformación correspondiente a muestra actual de animación, en el caso en el que la malla esté animada por transformación.

Si la malla no está animada, pero depende de una articulación, se le aplica la traslación en reposo. De lo contrario, la malla no quedaría colocada correctamente en su posición.

En el caso en el que la malla dependa de una articulación con *billboard* asignado, éste es aplicado. Pare ello, dependiendo del tipo de *billboard* (esférico o cilíndrico), se realiza una operación u otra. En ambos casos, el proceso conlleva operar sobre la matriz de modelo (*modelview*).

La Nintendo DS no permite extraer dicha matriz para operar directamente sobre ella, pero sí la matriz de clip, que es la multiplicación de la matriz de modelo y la de proyección. Para extraer únicamente la primera, basta con cargar previamente la matriz de proyección con la matriz identidad (guardando la matriz que hubiera antes con *glPush-Matrix*() y *glPopMatrix*()) y seguidamente extraer la matriz de clip, cargada en ese momento únicamente con la matriz de modelo. Esta operación la realiza internamente Libnds al solicitar la matriz de posición mediante *glGetFixed*(), insertando como parámetro GL_GET_MATRIX_POSITION y el array de 16 enteros donde se va a almacenar la matriz. Hay que tener cuidado con esta función, ya que deja el modo de matriz en GL_PROYECTION, por lo que hay que cambiar al modo GL_MODELVIEW para continuar con la transformación.

Una vez obtenida la matriz de modelo, para aplicar el *billboard* esférico, basta con cargar los valores correspondientes a la rotación con la matriz identidad. Esto es:

$$\begin{bmatrix} 1 & 0 & 0 & m[3] \\ 0 & 1 & 0 & m[7] \\ 0 & 0 & 1 & m[11] \\ m[12] & m[13] & [14] & m[15] \end{bmatrix}$$

De la misma forma, para aplicar el *billboard* cilíndrico, se elimina la rotación a derechas y frontal y se mantiene la rotación cenital. Esto es:

$$\begin{bmatrix} 1 & m[1] & 0 & m[3] \\ 0 & m[5] & 0 & m[7] \\ 0 & m[9] & 1 & m[11] \\ m[12] & m[13] & [14] & m[15] \end{bmatrix}$$

Si la malla no depende de ninguna articulación, se mandan los vértices sin aplicarles ninguna transformación.

int get_mesh_properties(edl *edlp, int meshN, int options)

Esta función extrae las propiedades de la malla considerada, cuyo índice viene definido por *meshN*. De esta forma, extrayendo sus atributos propios almacenados en el modelo edl y teniendo en cuenta las opciones de renderizado (*options*), se devuelve un entero con las propiedades que han de ser aplicadas a la malla, bajo una estructura de banderas de bit.

void cull_mesh(edl *edlp, int meshN, u32 * poly_attributes)

Esta función determina si hay que descartar las caras traseras de una malla de índice *meshN*, por ser ésta un objeto sólido, o por el contrario hay que mantenerlas. Para ello, se accede a las banderas de dicha malla del modelo edl y se identifica dicha característica. Si hay que hacer *backface culling* se le suma el valor de POLY_CULL_BACK a *poly_attributes* (que es el parámetro pasado como referencia de aquel que se empleará para determinar los atributos de la malla mediante *glPolyFmt*()). De lo contrario se le suma POLY_CULL_NONE.

void send_mesh_material(edl_instance *edlip, int meshN, int mesh_properties, render_pass_type type)

En esta función se asigna el material de la malla cuyo índice viene definido por *meshN*, teniendo en cuenta las propiedades de ésta, previamente extraídas con *get_mesh_properties*(), y el tipo de pasada de render. De esta forma, si la malla dispone de material, éste es aplicado, si la pasada de render es la principal se activa su textura (si dicho material tiene), si es una pasada de segunda textura ésta es activada y si es una pasada de reflexión, se activa la textura correspondiente a ésta.

void send_vertices(edl_instance * edlip, int meshN, int mesh_properties, render_pass_type type)

Esta función envía los vértices de la malla considerada, cuyo índice viene dado por *meshN*, al motor de geometría del hardware. Para ello se manda el comando de inicio de envío de vértices mediante la llamada a *glBegin*(), para posteriormente enviar los datos propios de los vértices. Así, se envían, en este orden, los colores por vértice (si así el parámetro *mesh_properties* indica de su incorporación), sus coordenadas de textura (si tiene), sus normales (si dispone de ellas) y finalmente sus coordenadas.

Estos datos se envían siguiendo el orden que indica la lista de triángulos de la malla, de tal forma que el hardware generará dichas primitivas bajo este orden.

Si la malla está animada por muestreo de vértices, y la animación está reproduciéndose, se envían los vértices y, en el caso en el que tenga, las normales correspondientes a la muestra de animación actual. De esta forma, para cada muestra se envían los vértices en posiciones distintas.

void edli_next_sample(edl_instance * edli)

Esta función actualiza las muestra de animación actuales de los diferentes esqueletos de la instancia del modelo edl especificada (*edli*). Con este fin, para cada esqueleto del modelo se tienen en cuenta una serie de parámetros:

- la **costumbre de animación** activa
- el **periodo de muestreo** de la animación
- si **la animación es cíclica** o se **repite un número determinado** de veces
- si la animación se está **reproduciendo**
- el **modo de reproducción** de la animación (forward, backward o ping-pong)

De esta forma, realizando las comprobaciones pertinentes, se actualiza, en cada llamada a la función, la muestra de animación actual de cada uno de los esqueletos del modelo. Por ello, esta función se llama desde la función *render_edl*(), ya que ha de realizarse este proceso una vez por ciclo (60 fps).

void edli_tex_anim_next_sample(edl_instance * edli)

Esta función actualiza las posiciones de las subimágenes, en función del tiempo, de los materiales con animación de textura del modelo edl especificado. El proceso tiene en cuenta el periodo de muestreo de la animación de textura y el número de subimágenes en horizontal y en vertical de la textura animada. De esta forma, según la velocidad de animación, se va recorriendo la textura de subimagen en subimagen, almacenando, para cada material con animación de textura, la posición de la subimagen actual.

Así, en cada frame de animación se muestra una subimagen distinta, simulando un efecto de movimiento, dado por la variación entre subimágenes.

6.4. Ejemplo de uso

Una vez detallada la librería msNDS y su funcionamiento básico, es conveniente mostrar el modo de empleo de ésta, ilustrándolo con un ejemplo de uso básico. Se explicará por lo tanto los procesos que se deben llevar a cabo obligatoriamente para el correcto funcionamiento del sistema de render, comunes a cualquier aplicación que se desarrolle con esta herramienta, y algunos ejemplos de utilización de las funciones de acceso, cuyo empleo dependerá de la aplicación en cuestión y sus requerimientos.

6.4.1. Diseño básico de la aplicación

Para la aplicación de ejemplo, considerar un personaje animado controlable por el usuario. Éste estará representado por un modelo con un único esqueleto y tres costumbres de animación: quieto (pequeños movimientos sutiles para aumentar la sensación de que el personaje es un ser vivo), correr y saltar. Así, cuando el jugador pulse los botones de dirección el personaje se moverá corriendo en dicha dirección, cuando pulse el botón B saltará, quedándose quieto cuando no se pulse ningún botón.

Además, este personaje puede equiparse con una espada, siendo ésta un modelo independiente. De esta forma, cuando se equipe la espada, ésta aparecerá en la mano del personaje.

Por otro lado, el escenario por el que se mueve el personaje se puede concebir como un modelo con casas y un plano como suelo. Dado que no se ha implementado un sistema de colisiones, tan solo el sistema de render, no es posible con esta herramienta impedir que los objetos se atraviesen.

6.4.2. Preparativos previos

El primer proceso necesario en este caso es el de creación de los recursos que se van a emplear. Para ello se modelan y texturizan el personaje principal, la espada y el escenario. Este proceso se realiza con MilkShape 3D, para poder exportar a MilkShape ASCII.

El personaje principal se anima mediante un esqueleto compuesto por varias articulaciones. Ha de crearse una articulación en el extremo de la mano, que aunque no anime nada, servirá para colocar la espada en ella. Además, a dicha articulación hay que asignarle los comentarios #EP y #ER, para indicarle al conversor EDL que han de exportarse sus transformaciones, para poder asignárselas más tarde a la espada.

Todos los modelos se exportan al formato de texto MilkShape ASCII, para a continuación ser convertidos con el conversor EDL a dicho formato, especificando la versión 3, que es la que incorpora todos los elementos del sistema de render.

Estos son los archivos que han de incorporarse al programa y cuyos datos se pasarán a la función *buil_edl*() para leerlos y generar los contenedores con los que trabaja el sistema de renderizado.

Además, como ya se ha explicado anteriormente, es necesario convertir las texturas empleadas en los modelos a cualquiera de los formatos que la Nintendo DS maneja, con Grit o con Nitro Texture Converter, a fin de poder emplearlos como texturas en el programa.

En la Figura 6.4 se muestra un esquema que resume este proceso previo.

6.4.3. Inicialización del sistema

El primer paso en la creación de la aplicación 3d es, comentado en el capítulo correspondiente a la librería, inicializar el sistema GL empleado por Libnds y los estados del hardware que se quieran activar, así como la memoria de video. De esta forma, se asigna el motor principal a una de las dos pantallas, se asignan los bancos de memoria de video necesarios para texturas y paletas (en este caso tan sencillo se pueden asignar todos los disponibles, dado que solo es necesario cargar texturas, sin embargo en otro caso donde haya fondos y sprites habrá que gestionar bien este aspecto) y se activan los estados del hardware necesarios para aplicar según qué efectos, como el anti-aliasing, el *blending*, la niebla, el sombreado *cartoon* o el marcado de contornos.

Antes de pasar a generar los modelos edl, es necesario inicializar el gestor de recursos. En este caso basta con inicializar la lista principal de recursos de tipo textura con *InitMainAssetList*() e insertar en ella las texturas correspondientes a los modelos empleados. Para ello hay que definir las texturas de tipo *texture_descriptor_t* correspondientes a cada

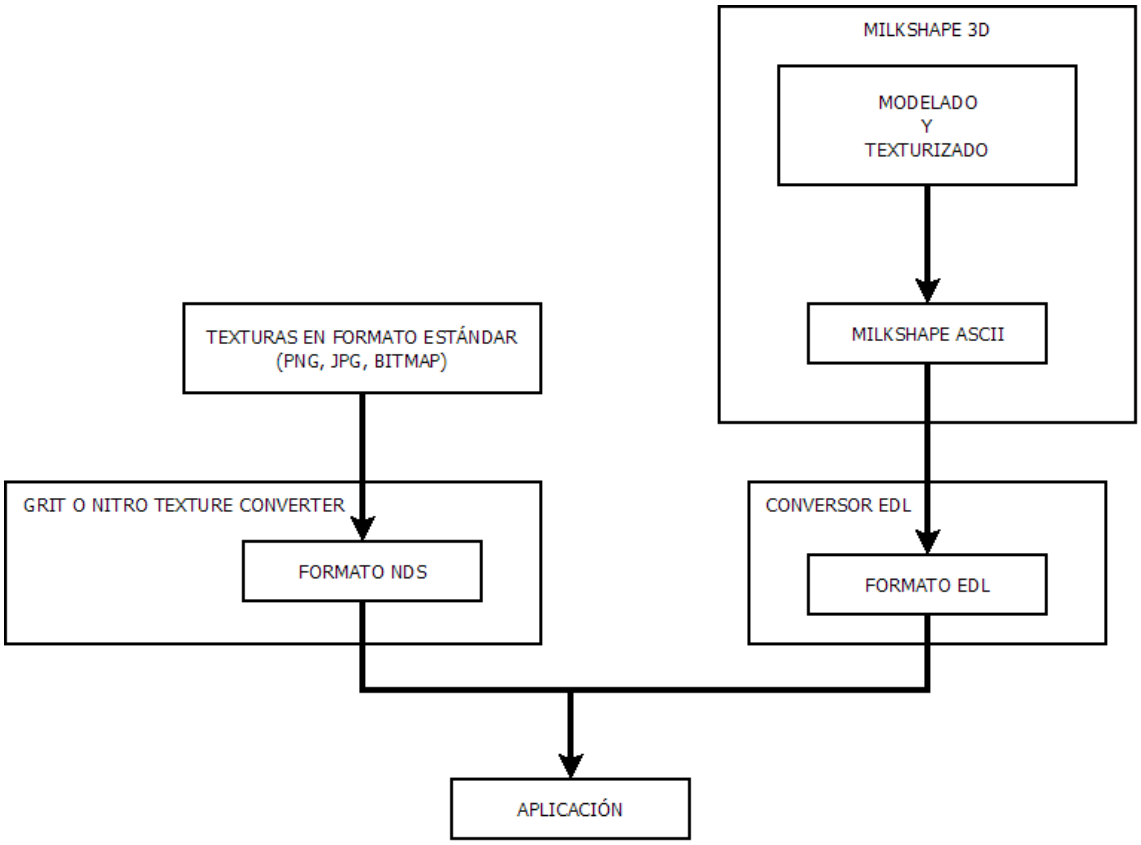

Figura 6.4: Proceso previo del ejemplo de aplicación propuesto.

una, especificando cada uno de los campos de ésta, como la dirección de los datos de la textura y su paleta incluidos en el programa, sus dimensiones, su tipo, su nombre y si se quiere el color de índice 0 transparente. Una vez definidas, se crean los recursos a partir de éstas, mediante la función *SetTextureAsset*(), que se almacenan en sendas variables de tipo ASSET_TEXTURE. Cada uno de estos recursos generados se introduce en la lista llamando a la función *AddTextureAssetToCurrentAssetList*(), pasándole el recurso que se quiere introducir en cada caso.

Habiendo incluido los datos de los modelos edl previamente generados, se pueden usar éstos para generar las estructuras con las que operar. Para ello, se definen tres variables de tipo edl y a cada una se le asigna el valor devuelto por *build_edl*(), llamada una vez por cada modelo, pasándole los datos incluidos de cada uno como parámetro de entrada. De esta forma, se dispone de los tres modelos edl que se pintarán posteriormente.

Sin embargo, no basta con definir estos modelos edl. Es necesario crear instancias de éstos (en este caso tan solo se requiere una por cada modelo), que son las que realmente maneja el sistema de render. Por lo tanto, se crean otras tres variables de tipo *edl_instance* para albergar los elementos de cada uno de los modelos, devueltos por la función *get_edl_instance*(), a la que se le pasa en cada caso el modelo edl correspondiente.

Para finalizar el proceso de inicialización de los modelos edl, se cargan las texturas correspondientes a cada uno (extrayéndolas de la lista de recursos), se convierten sus coordenadas de textura y se generan y activan, mediante las funciones *load_model_textures*(), *convert_texture_coordinates*() y *bind_model_textures*() respectivamente, pasándoles cada uno de los modelos edl (no sus instancias) en cada caso.

Como último paso para la configuración inicial del sistema, han de definirse los parámetros para albergar las opciones de renderizado. Se pueden definir varios de éstos, uno por cada modelo, de forma que el pintado tendrá unas propiedades distintas en cada uno, o bien definir un único parámetro y asignarlo a cada modelo en el pintado, de forma que todos tengan siempre las mismas opciones de renderizado.

En la Figura 6.5 se muestra un esquema que resume este proceso de inicialización.

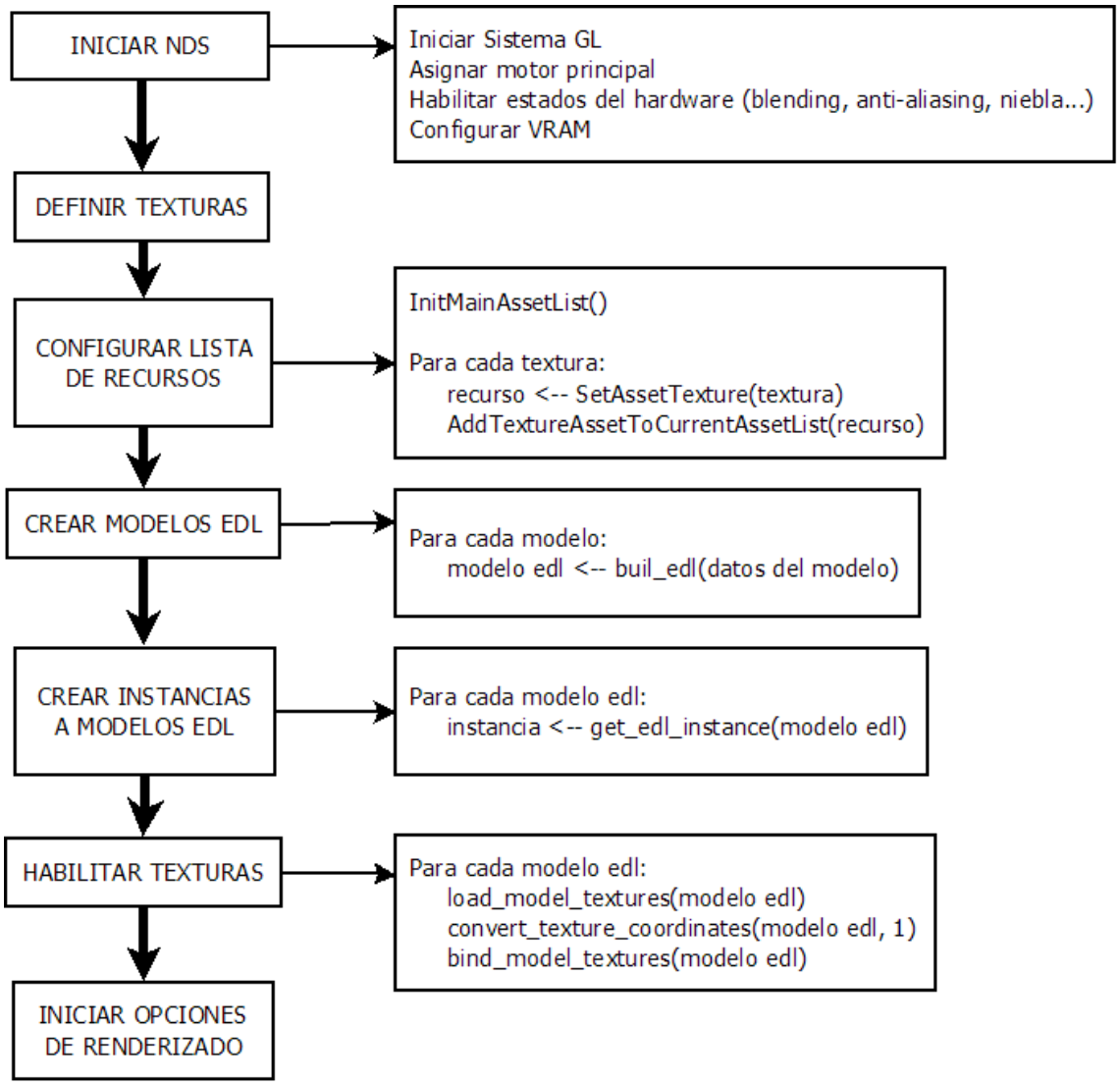

Figura 6.5: Proceso de inicialización del ejemplo de aplicación propuesto.

6.4.4. Implementación del sistema de juego

Finalmente, en el bucle principal de la aplicación, se implementa el sistema de movimiento del personaje en función de la interacción del usuario. De esta forma, según las acciones de éste sobre el los botones de la consola, se realizarán unas operaciones u otras. Así, se podrá aplicar la transformación correspondiente al modelo del personaje, además de establecer la animación correcta, para finalmente pintarlo en pantalla. Para finalizar, se aplicará la transformación correspondiente a la articulación de la mano del robot para

que, cuando la espada esté equipada, ésta se pinte en dicha posición. Un esquema de este proceso viene ilustrado de forma detallada en la Figura 6.6.

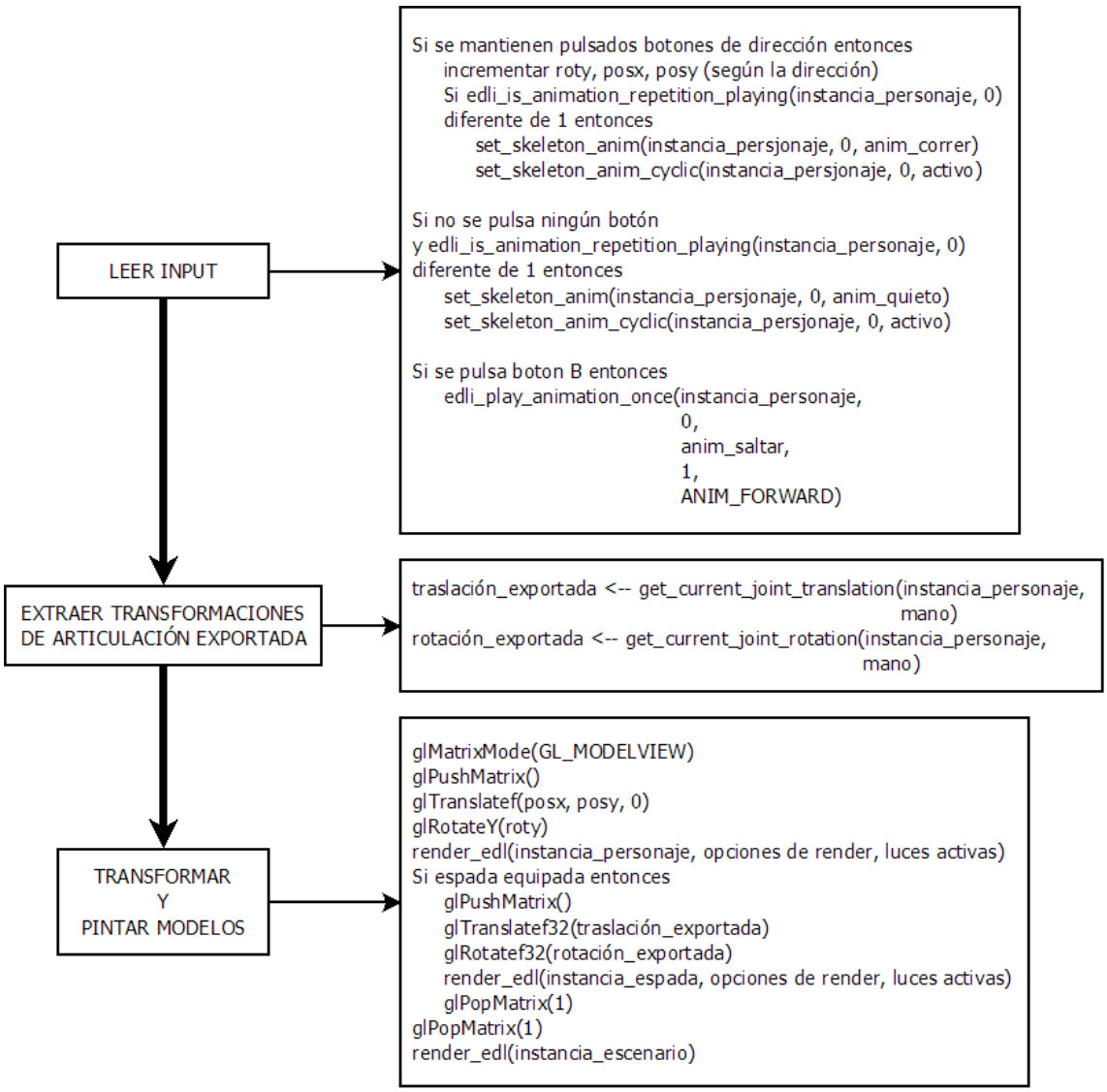

Figura 6.6: Implementación del sistema de juego del ejemplo de aplicación propuesto.

En este caso, se requiere que cuando se pulse cualquiera de los botones de dirección, se actualice la posición y la rotación del personaje en función de dicha dirección, así como cuando se pulse el botón B, se lance la animación de salto del personaje. Para ello, como se comenta en el capítulo referente a la librería Libnds, se llama a la función *scanKeys*() y se almacena en variables enteras el valor devuelto por *keysCurrent*(), *keysHeld*(), *keysDown*() o *keysUp*() para controlar en cada ciclo el estado de los botones del pad.

Así, cuando se pulsan los botones de dirección, se actualizan las variables donde se almacenan la posición y la rotación del modelo. Además, cuando se cumple este caso, se lanza la animación de correr mediante la función *set_skeleton_anim*(), en conjunción de *set_skeleton_anim_cyclic*() para que se reproduzca cíclicamente mientras se pulsan los botones de dirección. Sin embargo, para que esto se realice hay que comprobar que no se esté reproduciendo la animación de saltar, mediante la función *edli_is_animation_repetition_playing*(), para no cortar la reproducción de ésta, que se lanza

mediante la función *edli_play_animation_once*() cuando se pulsa el botón B. En el caso de no pulsar ningún botón, de la misma forma que se lanza la animación de correr, se lanza la de quieto.

Una vez controladas las transformaciones en el espacio y las animaciones del personaje, hay que extraer la traslación y la rotación de la mano del personaje en cada muestra de animación, para posicionar en este lugar la espada. Esto se realiza asignando a unas variables *edl_translation* y *edl_rotation* los valores devueltos por las funciones *get_current_joint_translation*() y *get_current_joint_rotation*() respectivamente. De esta forma, en cada ciclo se conoce la transformación de la articulación para aplicársela a la espada.

La última etapa consiste en aplicar las transformaciones a los modelos para moverlos por el escenario, seguido del pintado de éstos. Como se muestra en la Figura 6.6, este proceso consiste en, por un lado, aplicar la transformación de lo que se va a pintar a continuación, seguido del propio pintado. Por lo tanto, para pintar y posicionar correctamente el modelo del personaje, se aplica la traslación y la rotación definidas por la interacción del usuario, con las funciones de Libnds *glTranslatef*() y *glRotatef*(), guardando previamente la matriz de modelo para que esta transformación no se aplique más que al modelo (y, como se verá más adelante, a la espada). Una vez definida esta transformación, se manda pintar el modelo con la función render_edl().

En el caso en el que se equipe la espada, ésta ha de pintarse en la mano del personaje. Para ello se han obtenido previamente la traslación y la rotación de la articulación en ese instante de tiempo. Por lo tanto, de igual forma que se ha aplicado la transformación al modelo del personaje, se aplica esta transformación a la espada. Sin embargo, hay que realizar esta operación dentro del guardado de matriz (entre *glPushMatrix*() y *glPopMatrix*()), ya que de no hacerlo, se aplicaría la transformación de la articulación de la mano en el origen, y la espada no seguiría el movimiento del personaje por el escenario. Tras aplicar dicha transformación solo sería necesario pintar el modelo de la espada.

Finalmente, tan solo resta pintar el escenario, exento de transformación, ya que éste queda inmóvil ante el movimiento del personaje.

Como se puede apreciar, para realizar una aplicación 3D básica, con un grado de complejidad relativamente alto (en lo que se refiere a la Nintendo DS), mediante el uso de la librería msNDS se facilita sustancialmente el trabajo, bastando unos pocos comandos para poner en marcha el sistema de renderizado y animación. Esto permite al desarrollador centrarse en la propia aplicación y su funcionamiento, rodeando la complejidad técnica que supone implementar un sistema de renderizado y animación propio.

Capítulo 7

Capacidades gráficas de la Nintendo DS

La implementación de una herramienta de renderizado ha permitido experimentar con la Nintendo DS y comprobar sus capacidades gráficas, dado que se ha tratado de sacar el máximo provecho de éstas.

Por lo tanto, este capítulo recapitula las posibilidades del motor 3D de la Nintendo DS, siendo éste el objetivo principal del proyecto, exponiendo los procesos que realiza el hardware y su *pipeline*, así como los efectos que se pueden conseguir mediante el software.

Se comentarán igualmente las limitaciones propias del chip gráfico de la consola y las barreras que se han encontrado en el desarrollo de la librería de renderizado en este contexto.

7.1. Posibilidades del hardware

EL chip gráfico de la Nintendo DS, al igual que todos los procesadores gráficos, trabaja internamente siguiendo un *pipeline* gráfico que está constituido por varios procesos, tal y como se explica en el capítulo correspondiente a los gráficos por ordenador. Dado que la arquitectura del chip gráfico de la Nintendo DS es propia, al igual que el resto de consolas anteriores a la incorporación de los shaders (esto es Playstation 3, Xbox 360 e incluso la actual consola portátil de Sony, Playstation Vita, y la sucesora de la Nintendo DS, la Nintendo 3DS), la forma en que se realizan estos procesos es específica de esta consola y no se puede modificar desde fuera (simplemente se pueden configurar ciertos parámetros). Por lo tanto, los procesos comunes al resto de chips gráficos (procesado de primitivas, iluminación, texturizado) siguen una estructura propia, con unos cálculos internos específicos de esta consola. Además, incorpora varios efectos que se diferencian de otras aceleradoras gráficas, tales como el sombreado toon interno y el resalte de contornos.

Según los conocimientos adquiridos a lo largo de la investigación de la plataforma y el desarrollo de la herramienta de render, se han extraído ciertas conclusiones acerca de las capacidades intrínsecas del hardware de la Nintendo DS, en las que se incluyen tanto sus limitaciones como las bondades que se pueden explotar.

7.1.1. Representación de primitivas

Las primitivas son los elementos geométricos con información espacial con los cuales se construye la geometría tridimensional, para posteriormente ser rasterizados, generando píxels en el buffer correspondiente a la imagen en pantalla. Su construcción se realiza uniendo los vértices que los componen. Según el número de vértices que se empleen, se hablará de una primitiva u otra.

Primitivas

El motor 3D de la Nintendo DS permite, al igual que el resto de procesadores gráficos, varios modos de representación de primitivas. Así, una superficie se construye a partir de varios polígonos que van definiendo su forma. Estos polígonos corresponden a las primitivas que, dependiendo de las necesidades del renderizador, tendrán una geometría u otra. Los tipos de primitiva que permite la Nintendo DS, representados en la Figura 7.1, son:

- **líneas**: esta primitiva no está específicamente contemplada en la Nintendo DS, sin embargo se puede conseguir definiendo un triángulo con dos de sus vértices superpuestos. Esta primitiva, ya que no ofrece un aspecto sólido, puede servir para realizar efectos visuales.
- **triángulos**: esta es la primitiva más común en las aplicaciones de renderizado, dado que es el plano más sencillo que hay (definido por tres puntos), pudiendo definirse cualquier superficie a base de triángulos. La Nintendo DS permite representar los triángulos de dos formas:
  - **triángulos independientes**: cada una de estas primitivas se define mediante tres vértices. Al generar un nuevo triángulo se han de especificar tres nuevos vértices, aunque algunos de éstos coincidan en ambos triángulos. Esta es la forma más sencilla de representar triángulos, ya que en la mayoría de formatos los polígonos que constituyen las mayas vienen definidos por tres vértices independientes, sin importar la posición relativa de los triángulos entre sí, por lo que simplemente hay que hay que lanzar estos tres vértices para componer la primitiva. Esta es la primitiva empleada en el renderizador msNDS para representar los polígonos, dado que en el formato MilkShape ASCII así se definen. El número de vértices definidos (que ocupan memoria de la Vertex RAM) es de 3xT (siendo T el número de triángulos representados).
  - **triángulos consecutivos**: en este caso los triángulos se definen teniendo en cuenta los vértices especificados para el triángulo anterior. Es decir que se emplean los dos últimos vértices definidos en el triángulo anterior más un nuevo vértice en la construcción de un triángulo nuevo. De esta forma, se va generando una superficie continua a base de triángulos encadenados, disminuyendo así el número de vértices que definir. En este caso, este número es 3 + (T - 1). Sin embargo, debido a que la mayoría de formatos de salida de los programas de modelado agrupan los vértices correspondientes a un triángulo para su formación, es preciso emplear un algoritmo que genera largas cadenas de *triangle strips* a partir de cualquier geometría. En este caso, empleando esta técnica se acelera considerablemente el proceso de renderizado. El renderizador

msNDS no emplea esta primitiva, siendo una posible mejora de cara a posibles ampliaciones futuras.

- **cuadriláteros o quads**: esta primitiva se define mediante cuatro vértices, generando así un cuadrilátero. Ésta se emplea menos debido a su mayor complejidad y a que puede causar errores si el orden de envío de los vértices no es correcto, provocando cruces, y por lo tanto imperfecciones, en la geometría del polígono. En este caso también se pueden definir de forma independiente o consecutiva:
  - **quads independientes**: en este modo, los *quads* se definen mediante cuatro vértices independientes. Aunque no se empleen para renderizar modelos cargados (ya que prácticamente la totalidad de los formatos no almacenan los polígonos de esta forma), puede resultar útil para efectos de partículas, en los que se emplean cuadriláteros coloreados o con texturas, con los que se representan efectos como el humo o el fuego. En este caso el número de vértices que hay que definir es 4xQ (siendo Q el número de quads representados).
  - **quads consecutivos**: esta forma de primitiva se define de la misma forma que los triángulos consecutivos, salvo que se emplean dos vértices del polígono anterior y dos nuevos vértices. Esta forma de primitiva tiene poca aplicación, si no es para la generación de geometría in situ, como terrenos y ciertos efectos, ya que en la gran matoría de formatos se representan las mallas mediante una sucesión de triángulos. En este caso el número de vértices definidos disminuye, siendo su valor 4+(N - 1)x2, ahorrando así espacio de la Vertex RAM.

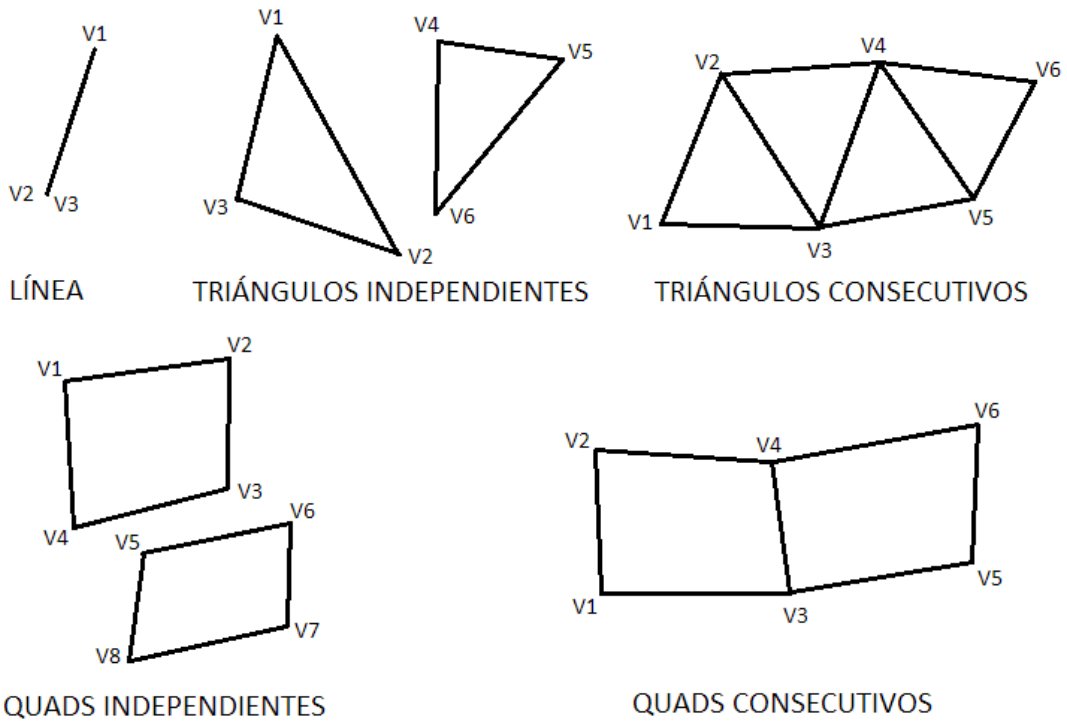

Figura 7.1: Primitivas con las que trabaja la Nintendo DS.

Precisión de los vértices

A la hora de enviar los vértices para la construcción de las primitivas, la Nintendo DS permite dos niveles de precisión de sus coordenadas. A menor precisión, menor resolución tendrá el modelo, dado que perderá detalle al no disponer de separaciones entre vértices menores. Esta precisión se adquiere mediante la profundidad de bit con la que se almacena cada una de las coordenadas del vértice.

Dado que el procesador de la Nintendo DS trabaja con números de punto fijo, se añade precisión a las coordenadas aumentando la parte fraccionaria de éstas. De esta forma, los dos formatos para enviar los vértices al hardware son:

- **V10**: 10 bits por coordenada, en formato 4.6 (6 bits para parte fraccionaria) con signo. Dado que las tres coordenadas se pueden almacenar en una única palabra de 32 bits, un vértice se puede enviar al hardware mediante un único parámetro en el que los diez primeros bits constituyen la coordenada x, los diez siguientes la coordenada y, y los siguientes diez bits corresponden a la coordenada z, quedando inutilizados los dos últimos bits del parámetro.
- **V16**: 16 bits por coordenada, en formato 4.12 (12 bits para parte fraccionaria) con signo. En esta caso, las tres coordenadas no se pueden almacenar en una única palabra de 32 bits, por lo que un vértice se ha enviar al hardware mediante dos parámetros de un mismo comando: en el primero se envía las coordenadas x e y, y en el siguiente la coordenada z, quedando inutilizados 16 bits de éste último.

Dado que ambos formatos disponen de 3 bits de parte entera (el último corresponde al signo), las coordenadas de los vértices tienen que estar comprendidas en el intervalo [-8, (8 - 2-12)], en el caso de emplear el formato V16 y [-8, (8 - 2-6)], por lo que en el formato de entrada del modelo hay que asegurarse de que esto se cumple.

Resulta evidente, que a mayor precisión, mayor gasto de memoria, por lo que es conveniente emplear el formato de menor precisión en aquellos modelos que necesiten menor resolución.

Backface culling

La Nintendo DS tiene una limitación en cuanto al número de vértices que se pueden representar en pantalla. Ésta es de 6144 vértices por frame, lo que se traducen en 2048 triángulos o 1535 cuadriláteros. Aunque en cualquier plataforma es importante ahorrar recursos y deshacerse de la geometría innecesaria (aquella que no se va a ver), esta limitación lo hace aún más importante en esta consola.

Una de las formas más comunes de ahorrar geometría es desechar las caras traseras de los objetos sólidos (de espaldas a la cámara), dado que siempre van a estar tapadas por las caras frontales del modelo. De esta forma, dependiendo del sentido en el que se manden pintar los vértices de las primitivas, el hardware reconoce si se trata de una cara trasera, o por el contrario está de frente a la cámara. Si los vértices de un polígono se pintan en sentido anti-horario, éste se considera frontal, de lo contrario se considera trasero. Generalmente este proceso se realiza en la fase de modelado, en la que el software ordena adecuadamente los vértices de los polígonos, según la dirección normal de éstos, a la hora de exportar sus datos.

Realizar el llamado *backface culling* (es decir, mandar al procesador de la consola que deseche estas caras traseras, proceso que realiza tras la transformación de vista, antes de digitalizar/rasterizar) es sumamente importante para poder aprovechar los recursos de la consola en un mayor número de modelos, en modelos más detallados o en realizar varias pasadas de renderizado.

Sin embargo, hay que tener en cuenta que si el modelo no es sólido (si alguna de sus superficies se ve por ambas caras, como una caja abierta) hay que desactivar este proceso. De lo contrarío las caras traseras no se ven y el modelo no se representa correctamente. Este efecto se puede ver en la Figura 7.2.

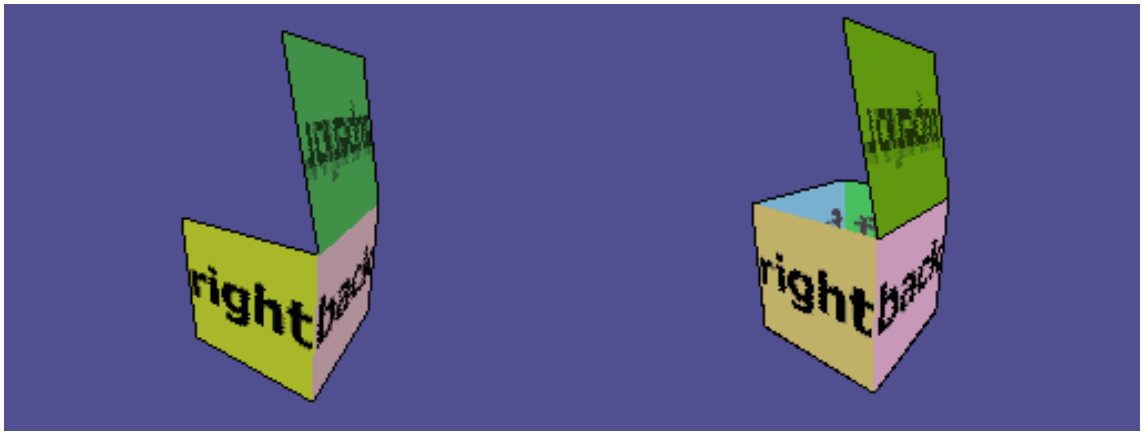

Figura 7.2: *Bacface culling* aplicado sobre un objeto no sólido. En la figura izquierda se puede comprobar que los polígonos que presentan su cara trasera se están desechando, mientras que en la figura derecha se pintan correctamente.

Clipping

Los objetos que se representan en pantalla son aquellos que están dentro del llamado volumen de vista (*view volume*) o frustrum. De esta forma se puede limitar la distancia de dibujado y controlar así el número de polígonos susceptibles de ser pintados.

En el caso en el que un polígono intersecte un lado del frustrum, se crean dos nuevos vértices en el límite del volumen para reemplazar los que están fuera de éste, proceso representado en la Figura 7.3. Este es un factor a tener en cuenta, ya que cuantos más vértices intersectados haya, mayor es el uso de la Vertex RAM (aunque un solo vértice se quede fuera, se generan otros dos para formar el polígono). En el caso en el que se quede todo el polígono fuera, éste es desechado por completo, por lo que con consumen memoria de la Vertex RAM ni de la Polygon RAM.

Por esta razón, la Nintendo DS da la opción de controlar este proceso: o bien se cortan los polígonos, con la generación de geometría adicional, o bien se dejan de pintar los polígonos que corten los límites del frustrum, ahorrando así memoria adicional, pero con un posible efecto dentado en la geometría del modelo.

Asimismo, el chip gráfico de la Nintendo DS permite, mediante la activación de un registro determinado, desechar los polígonos muy pequeños, los cuales se representarían con un único píxel. De esta forma se limita un poco más la geometría representada en pantalla.

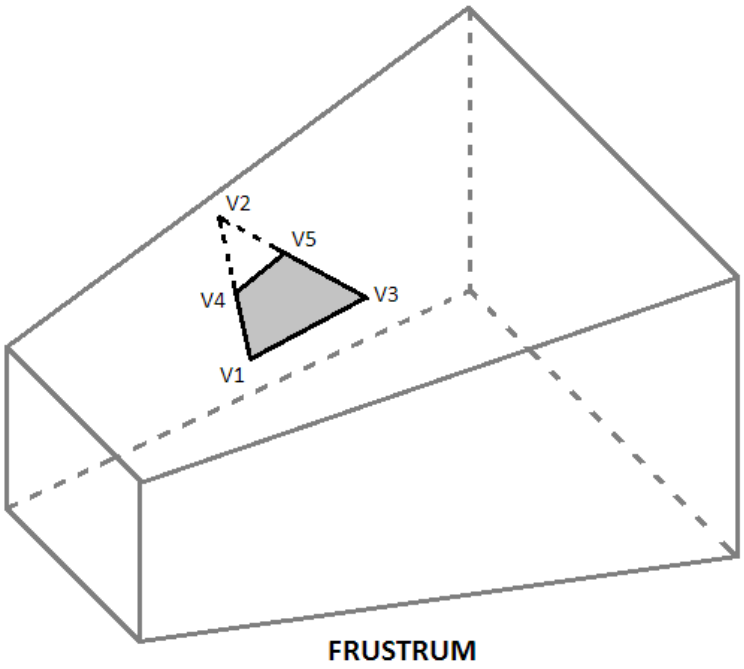

Figura 7.3: Proceso de *clipping* de un triángulo cortado por el frustrum.

7.1.2. Materiales, iluminación y color por vértice

La apariencia básica de un modelo se define en un primer momento mediante los materiales que le aportan color y un modelo de reflexión de la luz. De esta forma, según los parámetros del material considerado, la superficie afectada por éste será más mate o más pulida y brillante, además de conferirle un color. Por lo tanto, el material de una malla hace que su superficie reaccione de forma particular a la iluminación.

La Nintendo DS realiza cálculos sencillos de iluminación a partir de los parámetros básicos del material que permite especificar. Se aplica el sombreado Gouraud, dónde se realizan cálculos de iluminación por vértice, por lo que la definición del sombreado y los brillos generados dependen de la resolución poligonal de la malla. Por lo tanto, no se calcula una iluminación compleja, y no se permite aplicar mapas para definir los diferentes parámetros de los materiales, pero se aporta una apariencia y un sombreado básicos que confieren volumen y cierto realismo al objeto, que se verán reforzados mediante el texturizado.

Materiales

La Nintendo DS permite definir cuatro parámetros básicos del material:

- **color ambiente**: color reflejado de forma indirecta ante iluminación tenue. Suele ser el mismo que el difuso o algo más apagado.
- **color difuso**: color reflejado ante iluminación directa. Es el que aporta el color básico del objeto.
- **color especular**: color del resalte o brillo generado sobre la superficie del objeto

ante iluminación directa. Suele ser blanco o grisáceo, aunque en materiales metalizados el color del brillo es importante para definir su apariencia.

- **color emisivo**: color que emite el propio objeto aún a falta de iluminación.

Cada uno de estos atributos se especifica con un color en formato RGB con 5 bits por componente. Los atributos ambiente y difuso se envían al hardware mediante un parámetro de 32 bits que alberga ambos colores, 15 bits cada uno, el primero en los 16 primeros bits y el segundo en los siguientes. De forma análoga, el color especular y el emisivo se envían al hardware en un único parámetro correspondiente a estos atributos.

La Nintendo DS ofrece un sistema complejo y poco intuitivo para asignar la amplitud del resalte especular (*shininess*), basado en la generación de una tabla de 32 valores que definen el nivel de este parámetro en función de la dirección normal al vértice considerado, a la dirección de la luz (para cada una de las cuatro luces disponibles) y a la dirección de enfoque de la cámara. De esta forma, se puede controlar la amplitud de los brillos generados en materiales especulares, confiriendo una apariencia más o menos pulida, diferenciando materiales metalizados de materiales de plástico (brillos mayores y menores respectivamente). Sin embargo, este método resulta poco intuitivo y complejo de manejar para cada material de la escena, por lo que en la mayoría de los casos se optará por no emplear esta tabla, con lo que el hardware empleará una por defecto con una variación de incremento lineal, o por emplear una tabla genérica para todos los materiales (Libnds dispone de una función que genera una).

Las propiedades materiales se aplican en el momento de efectuar los cálculos de iluminación, es decir, cuando se envía el comando de normales, con el que definir el sombreado, y por lo tanto el color final en cada punto de la superficie.

Iluminación

La iluminación en la Nintendo DS es limitada, disponiendo únicamente de cuatro fuentes luminosas direccionales para iluminar la escena. Por lo tanto, a cada una de estas fuentes se les asigna una dirección y un color, parámetros que se emplearán en los cálculos de iluminación para cada objeto.

Al tratarse de luces direccionales (de rayos paralelos e infinitos), el hecho de que el número de éstas sea pequeño no es tan relevante, ya que con una única fuente se ilumina toda la escena con rayos en una determinada dirección (generando sombreado en consecuencia sobre los objetos). Por lo tanto, con cuatro fuentes se puede generar iluminación en cuatro direcciones y con cuatro colores diferentes, resultando suficiente en la mayoría de casos. De hecho, un número creciente de luces empleadas en la escena ralentiza el proceso de renderizado de forma importante.

Sin embargo, se puede simular iluminación puntual (los rayos se emiten desde un punta a todas las direcciones) variando la dirección de una fuente luminosa para cada polígono u objeto de la escena. De esta forma se puede emular que la luz se emite desde un mismo punto con direcciones diferentes.

La iluminación en la Nintendo DS tan solo genera sombreado sobre la superficie de los modelos, en función de los materiales y los vectores normales a ésta, como se puede apreciar en la Figura 7.4. En ningún caso proyecta sombras sobre otros objetos. Este efecto se puede conseguir por software, mediante algoritmos muy costosos de iluminación

dinámica, poco recomendables para un procesador tan limitado como el de la Nintendo DS. De igual forma, los objetos no reflejan luz sobre otros objetos, tan solo hacia la cámara (definiendo así el color con el que se ven).

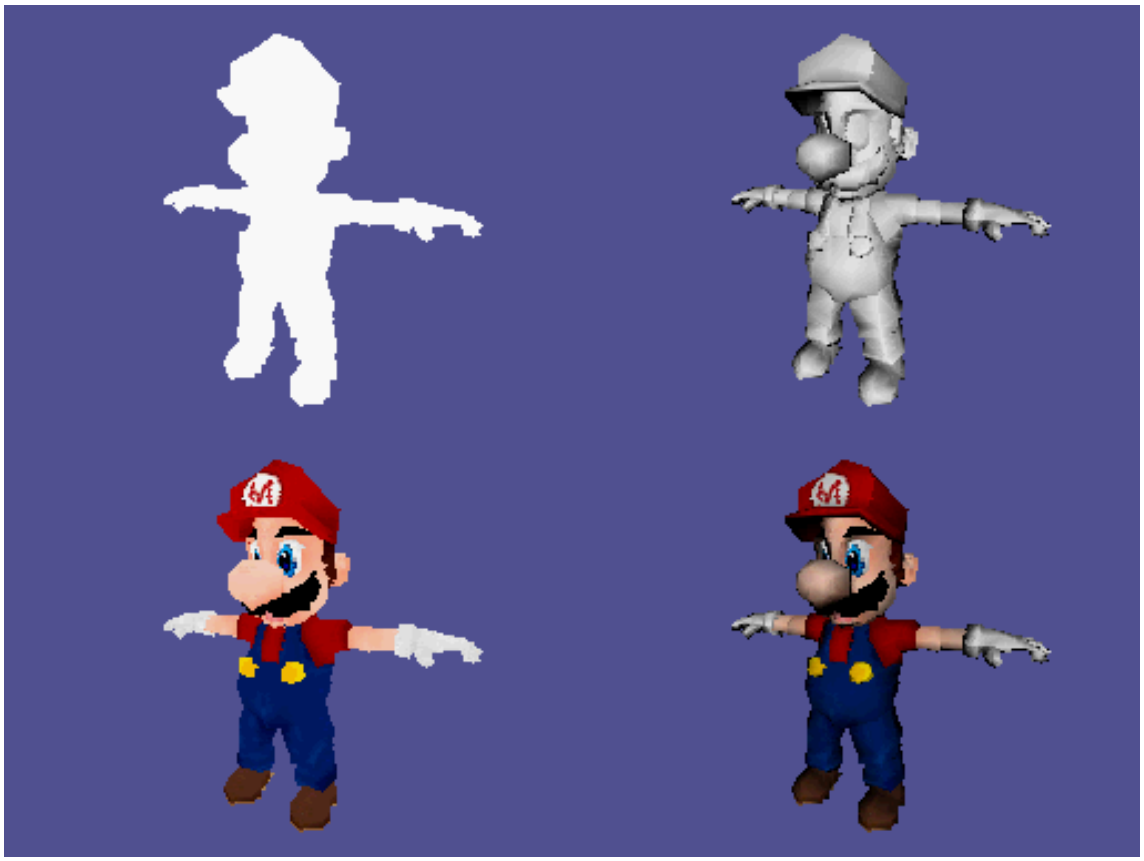

Figura 7.4: Iluminación sobre un modelo. En la primera figura no hay iluminación, aplicándose únicamente el material (blanco). En la siguiente, se aplica iluminación mediante el envío de normales, lo que da volumen al objeto. En la siguiente figura se ha aplicado la textura, quedando las normales desactivadas. Finalmente, en la última figura se realiza tanto el proceso de iluminación como de texturizado.

Para generar la sombra de un objeto en el plano del suelo hay varias formas de hacerlo. Una de ellas es por hardware, mediante los denominados volúmenes de sombra: se especifica un volumen (generalmente el modelo que proyecta la sombra) y, mediante dos pasadas al hardware, una que genera una máscara en el *stencil buffer* y otra que renderiza el volumen, con los polígonos de sombra correspondientes, se genera este efecto de sombra que vería en función de la posición del modelo y de la dirección de la luz.

El otro método (escalado a cero en vertical), mediante software, consiste en pintar el modelo que proyecta la sombra con todos los vértices en un mismo plano (el del suelo) y aplicarles un mismo color a todos ellos (normalmente negro o gris oscuro). De esta forma se crea una sombra con la forma del objeto. Sin embargo, este método no considera la dirección de la luz, por lo que no es un efecto físicamente realista, pero sí cumple la función de sombreado a relativo bajo coste.

Una mejora del método anterior es calcular la transformación de proyección sobre un plano del modelo. Esto puede hacerse tanto para luz de rayos paralelos como para una luz puntual, mejorando el efecto de iluminación dinámica.

Para sombras estáticas, de un escenario por ejemplo, se puede emplear el color por vértice para sombrear los polígonos correspondientes.

Transparencia

La Nintendo DS permite transparencia en polígonos con una resolución de 31 niveles. El valor 31 corresponde al opaco absoluto, mientras que el valor 1 es la mayor transparencia posible, sin llegar a ser completamente translúcido. El valor 0 de transparencia está reservado para pintar el polígono en modo alambre o wireframe, en el que únicamente se pintan las aristas de éste.

Para realizar correctamente la transparencia, es necesario pintar primero los polígonos opacos para asegurarse de que aquellos que se encuentran a mayor profundidad se almacenen correctamente en el *depth buffer*, y por lo tanto se pinten detrás de los objetos translúcidos, apreciando la superposición de ambos. A continuación se han de pintar los objetos con transparencia, sin modificar los valores de profundidad almacenados anteriormente en el *depth buffer* (para ello se ha de mantener desactivado el parámetro de valor de profundidad para pixels translúcidos del registro de atributos del polígono, de lo contrario almacenaría la nueva profundidad, impidiendo la visibilidad de los objetos opacos de detrás).

En la Figura 7.5 se aprecia este efecto.

Figura 7.5: Efecto de transparencia. Se puede apreciar cómo la visera del robot tiene componente de transparencia, permitiendo la visualización de la cabeza situada más atrás.

Color por vértice

Además de aplicar propiedades materiales a la malla, se puede definir de forma directa el color de los vértices, prescindiendo en este caso de la iluminación.

La técnica del color por vértice consiste en asignar un color a cada vértice de la malla en función de una referencia dada, afectando a los triángulos a los que pertenece el vértice (el color en el interior del polígono se interpola mediante los colores de los vértices que lo componen).

El vértice se representará mediante el color exacto especificado, obviando cálculos de iluminación. Por ello, una malla representada únicamente de esta forma no tendrá apa-

riencia de volumen, puesto que ésta la determinan los materiales en conjunción que la información de normales correspondientes.

De esta forma, se puede emular la apariencia de una textura, perdiendo evidentemente el detalle entre vértices que ofrece el texturizado, pero ahorrando memoria de vídeo. Puede usarse también como refuerzo al material y al texturizado para aportar un efecto adicional, dado que el color por vértice, el color material y el correspondiente a la textura se mezclan en la etapa de rasterizado.

En la Figura 7.6 se pueden apreciar los distintos efectos que se pueden conseguir con el color por vértice.

Figura 7.6: Distintas posibilidades del color por vértice. En las dos primeras figuras se muestra como, mediante el color por vértice se puede simular una textura, ahorrando memoria de video. En las dos figuras de abajo, se muestra la aplicación del color por vértice a un modelo texturizado para añadir una capa adicional a la apariencia del modelo, pudiendo simular iluminación o cualquier otro detalle extra.

En un apartado posterior se detalla cómo se pueden combinar el color del material, el color por vértice y el texturizado para dar una apariencia más detallada a la malla.

7.1.3. Texturizado

El texturizado es el último proceso de establecimiento de la apariencia de un modelo. En éste, como ya se ha comentado, se "pega" una imagen sobre la malla correspondiente, según las coordenadas de textura establecidas en el mapeado y asociadas a cada vértice,

o bien calculadas en tiempo de ejecución.

En la Nintendo DS el proceso de texturizado es de los que más recursos consume, dado que hay que almacenar las texturas que se van a emplear o activar en la memoria de vídeo, y ésta es muy escasa. Con el fin de ahorrar memoria se pueden almacenar las texturas en formatos indexados menos pesados a los que se les asocia una paleta de colores, cuyo peso depende del número de estos. De esta forma, para cada texel (tesela) no se almacena un color, sino un índice a unos de los colores almacenados en la paleta.

La consola trabaja con texturas de al menos 8x8 píxels y como máximo de 1024x1024 píxels.

Formatos de textura

Como ya se ha explicado en capítulos anteriores, la Nintendo DS trabaja con varios formatos de textura. Cada uno de ellos presenta unas características y un nivel de precisión propios. Éstos son:

- **RGB4**: 2 bits por texel, paleta asociada de 4 colores (16 bits por color).
- **RGB16**: 4 bits por texel, paleta asociada de 16 colores (16 bits por color).
- **RGB256**: 8 bits por texel, paleta asociada de 256 colores (16 bits por color).
- **RGB**: 16 bits por texel (5 bits por componente rgb más un bit de transparencia), color directo (sin paleta asociada).
- **A3I5**: 5 bits de color más 3 bits de transparencia por texel, paleta asociada de 32 colores (16 bits por color), 8 niveles de transparencia.
- **A5I3**: 3 bits de color más 5 bits de transparencia por texel, paleta asociada de 8 colores (16 bits por color), 32 niveles de transparencia.
- **Textura comprimida**: se comprime la textura agrupando bloques de 4x4 texels y codificando su contenido.

Para aplicar transparencia, a parte de los formatos A3I5 y A5I3 que presentan un canal de alfa con varios niveles de precisión y el formato RGB que dispone de un bit para definir esta propiedad, se puede establecer para el resto de formatos con paleta asociada que el índice 0 de ésta se considere completamente transparente. De esta forma, todos los texels con este color se harán transparentes. Esto puede ser útil para establecer una forma compleja, definida por el dibujo de la imagen, empleando únicamente un plano, ahorrando así geometría. Simplemente, en la fase de creación de la textura hay que tener en cuenta que este color se hará transparente y asociarlo al índice 0 de la paleta.

Repetición y volteo de texturas

Para mejorar la distribución de una textura y su resolución sobre la superficie del modelo, la Nintendo DS (como suelen hacerlo también muchos otros chips gráficos) permite repetir la textura a lo largo de los ejes x e y, de forma que con una textura más pequeña se puede cubrir mayor superficie de la malla. De esta forma se puede aumentas la resolución de la textura sobre el modelo sin aumentar su tamaño, y por lo tanto reducir el consumo de memoria de vídeo. Este efecto es útil en superficies regulares y repetitivas, como muros de ladrillos o suelos de baldosas.

En el caso en el que no se disponga de una textura repetible o "tileable" (del inglés *tile*, tesela), la consola también permite, en cada repetición, voltear la textura en la dirección de repetición, de forma que se produce un efecto espejo, en el que los bordes de las imágenes repetidas y volteadas coinciden perfectamente. De esta forma se evita la aparición de discontinuidades entre cada repetición, mejorando el efecto.

En el caso en el que la textura no cubra toda la superficie del modelo y no se active la repetición, se mantienen los texels de los bordes de la textura a lo largo de la superficie del modelo.

En la Figura 7.7 se aprecian estos diferentes procesos de texturizado.

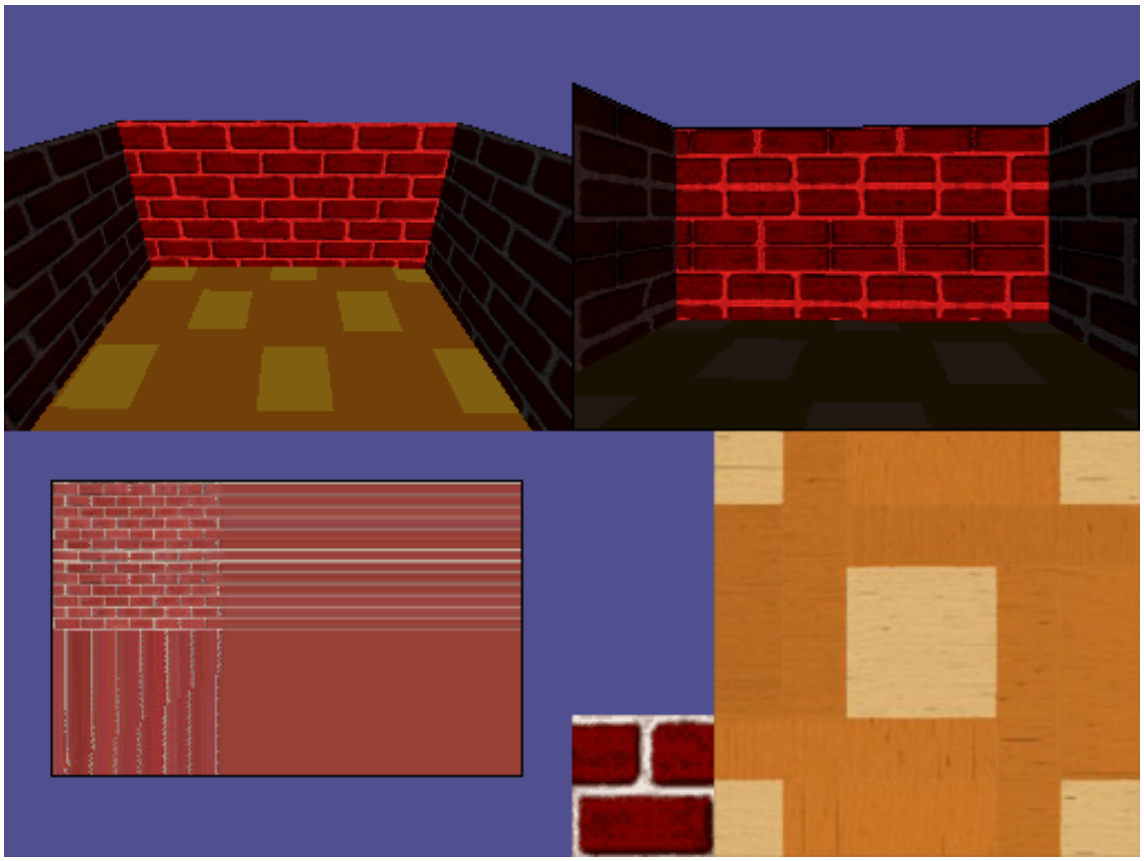

Figura 7.7: Repetición, volteo y "mantenimiento" de textura. En la primera figura se muestra la repetición de texturas mediante las imágenes teselables de la esquina inferior derecha. Se puede apreciar cómo, con una imagen reducida se puede texturizar una sección amplia del modelo, ahorrando memoria de video. En la siguiente figura se aprecia el efecto de volteo sobre la pared de ladrillo. En la última figura se aprecia cómo se mantienen los píxels de los bordes de la textura a lo largo de la superficie del modelo.

Mezcla

En la fase de rasterizado, en la que cada polígono se digitaliza, pasándolo a información bidimensional para formar la imagen de pantalla, es donde se aplica la textura y se mezcla con el color del polígono previamente enviado al hardware. En la Figura 7.8 se muestra la diferencia entre realizar no la mezcla.

Figura 7.8: Mezcla de transparencia aplicada (izquierda) y no (derecha) sobre un modelo. En la figura de la izquierda se puede apreciar cómo se mezclan los diferentes colores enviados al hardware para cada pixel en sucesivas capas de texturizado. Sin embargo, en la figura de la derecha únicamente se visualiza la última textura aplicada al modelo, ya que no se realiza la mezcla con los colores inferiores.

Básicamente hay dos modos de mezcla de textura (modulación y "calcomanía" o decal), aunque en el chip gráfico, conjuntamente con estos modos y mediante el mismo parámetro, se pueden especificar también los sombreados *toon* y *highlight*. Sin embargo, dado que el procesado de mezcla de textura en estos dos modos de sombreado es el mismo que en el modo modulación y lo que cambia es el tratamiento que se le hace al color del vértice, se comentarán más adelante y de forma específica estos modos de sombreado, ya que se trata de efectos gráficos característicos de la Nintendo DS.

Por lo tanto, los dos modos de mezcla de textura básicos implementados en la Nintendo DS son:

- **modulación**: en este modo de mezcla, se modulan de forma lineal el color del texel y el color del polígono (por material o por color por píxel), así como la transparencia en cada caso, correspondientes al fragmento considerado. Ambas componentes se procesan con el mismo peso, de forma que el resultado es la multiplicación de ambos, manteniendo el valor resultante en el intervalo de color. Así, el color de la textura y el del material se mezclan de forma homogénea.
- **"calcomanía" o *decal***: en este modo se emplea la transparencia de la textura (en caso de que el formato de textura disponga de este parámetro, sino se considera transparencia nula o valor de alfa 31) para modular los colores del polígono y del texel correspondiente al fragmento considerado, de forma que cuanto menor sea la transparencia (mayor el alfa) mayor sea el peso de la componente de color de la textura y menor la del color del polígono. De esta forma, en las zonas transparentes de la textura, predominará el color del material asignado al polígono (o su color directo) y en las zonas sólidas se mantendrá el color de la textura. De ahí su nombre, ya que el resultado se asemeja a pegar una pegatina sobre un objeto, en vez de quedar ambos colores mezclados. Se pueden extraer dos inconvenientes de este método: el primero es que se pierde el valor de transparencia de la textura, ya que solo se emplea como proporción del color de textura frente al del vértice, impidiendo utilizar este parámetro para determinar la transparencia global de la malla (tal y como se aprecia en la Figura 7.9). El otro inconveniente es que si la textura tiene

transparencia nula (alfa máximo), el color del vértice no se emplea, perdiendo la información de sombreado calculada mediante las normales, por lo que el modelo quedará sin volumen en las zonas donde esté aplicada dicha textura.

En la Figura 7.9 se muestra la diferencia entre los modos de mezcla de modulación y "calcomanía".

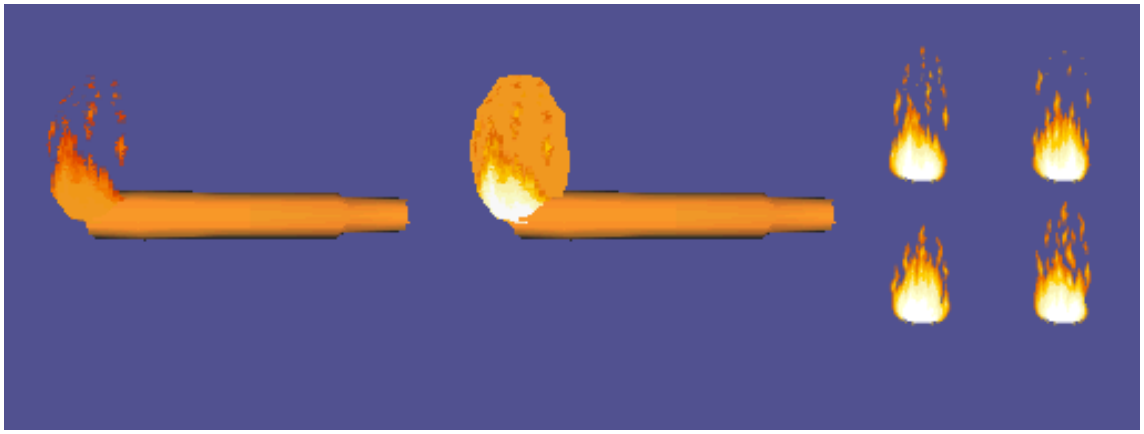

Figura 7.9: Modos de mezcla de modulación (izquierda) y "calcomanía" (centro). En el modo *decal* se comprueba que allí donde en la textura (derecha) hay transparencia, se visualiza el material de la malla, obviando dicha transparencia (en el modo modulación se emplea para dar transparencia a la propia malla), mientras que donde es opaca, se mantiene el color de la textura y no se mezcla con el material (como sí se hace en el modo modulación).

Modos de transformación de coordenadas de textura

La Nintendo DS dispone de tres modos de transformación de coordenadas de textura, con las cuales poder realizar transformaciones. De esta forma se pueden conseguir efectos como el flujo de agua de un río sin mover geometría o simular reflexión esférica.

Cada uno de estos modos de transformación se define por su fuente, a partir de la cual genera las coordenadas de textura que se van a transformar mediante la matriz de textura. Éstas son:

- **coordenadas de textura**: se emplean las coordenadas de textura enviadas con el comando correspondiente multiplicadas por la matriz de textura. De esta forma se pueden aplicar rotaciones, traslaciones y escalados de texturas.
- **normales**: se emplean las normales de la malla a la que se aplica la textura para generar las coordenadas de textura. De esta forma se traslada la deformación de la superficie del objeto a las coordenadas de textura, generando un efecto de reflexión esférica. Para que este efecto se aplique se han de enviar las susodichas normales. Adicionalmente, hay que realizar la traslación de la textura sobre el objeto para que ésta mire siempre de frente a la cámara y se mantenga el efecto de que un entorno se está reflejando sobre el objeto.
- **vértices**: se emplean las coordenadas de los vértices de la malla para producir desplazamientos de textura dependientes de las coordenadas de vista.

7.1.4. Niebla

Este es uno de los efectos propios realizados por hardware. Éste aplica una capa de color a los modelos afectados que varía según con profundidad. De esta forma, si el color de la niebla es el mismo que el del fondo, el color del modelo se va difuminando con el fondo, dando la sensación de lejanía y profundidad.

Este efecto es también útil para disimular los polígonos cortados por el frustrum, dado que estos se difuminan y se aprecia menos este hecho.

Se pueden configurar parámetros de la niebla (y por supuesto activar o desactivar el efecto) como su color (5 bits por componente rgb), su transparencia (5 bits), a partir de qué profundidad empezar a aplicar el efecto (15 bits), y una tabla de dispersión de la densidad de la niebla en función de la profundidad (32 valores). Se puede emplear un color específico (el que se ha comentado previamente) o simplemente utilizar la transparencia, empleando el color de fondo.

7.1.5. Resalte de contornos

Otro de los efectos propios y característicos de la Nintendo DS es el de resaltar los contornos de grupos de polígonos. Éste puede usarse como parte del interfaz de usuario para resaltar objetos seleccionados o simplemente como efecto gráfico para darle al apartado visual un aspecto de cómic.

Se pueden establecer hasta 8 colores de contornos, los cuales actúan cada uno sobre 8 identificadores de polígonos (hay 64 en total). De esta forma, para que un color de contorno actúe sobre un grupo de polígonos, éstos deben tener asociados unos de los 8 identificadores de polígono correspondientes a dicho color de contorno. Como siempre, cada color se define por tres componentes RGB de 5 bits cada una. En la Figura 7.10 se aprecia este efecto.

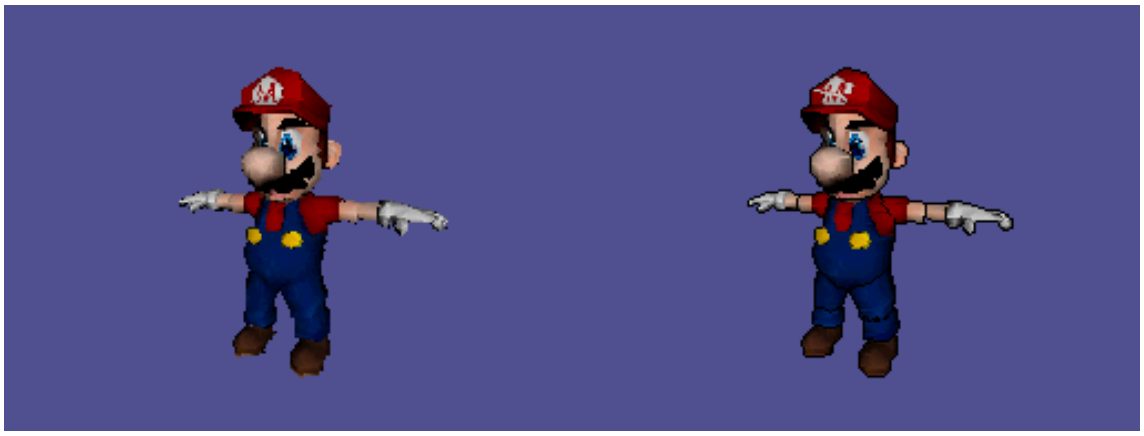

Figura 7.10: Contorno negro aplicado a un modelo.

7.1.6. Sombreado toon y highlight

Otro efecto característico de la consola que aplica un sombreado de pocos niveles de variación de color que otorga al apartado visual un estilo de cómic muy marcado.

El efecto consiste en utilizar pocos niveles de sombras en vez de una gama suavizada, tal y como se usa en el modo de mezcla de modulación, de forma que la diferencia entre los colores que corresponden a los diferentes niveles es muy notoria, generando este aspecto tan característico de los dibujos animados.

Para configurar este efecto es necesario establecer una tabla 32 colores (como máximo) asociados a los 32 niveles de rojo posibles. De esta forma, para cada píxel considerado en la etapa de rasterizado se extrae el nivel de rojo del polígono (que puede ser del material o del color directo del vértice) y se introduce en la tabla, que entrega nuevos valores de rojo, azul y verde, según el color que se haya definido para este valor. Finalmente se aplica la mezcla con la textura de la misma forma que en el caso del modo modulación, considerando el color del sombreado toon en vez del color del polígono (todo este proceso, salvo la definición de la tabla, es efectuado internamente por el procesador gráfico de la consola).

Este sistema parece poco utilizable, ya que los colores de salida de la tabla dependen de un valor de rojo, resultando poco intuitivo a la hora de definir la tabla (un nivel de rojo de 1 puede asociarse a un azul y un valor de 2 a un verde, no dependiendo en ningún caso del color real del fragmento calculado anteriormente).

Sin embargo, en combinación con una textura sí se puede aplicar un efecto de sombreado cartoon: el nivel de brillo del color del material calculado a partir de las normales determina el nivel de rojo sobre la superficie del modelo (quedando más bajo en zonas sombreadas y más alto en zonas donde incide directamente la luz). Definiendo niveles de gris como colores de salida de la tabla se puede simular un sombreado de pocos niveles ejercido por una luz blanca, ya que se generarán sombras grises. En el caso de querer generar el efecto de una luz azulada, habrá que asociar a los niveles de rojo de la tabla niveles de azul.

De esta forma, tras el mezclado de este sombreado con la textura se consigue la apariencia del modelo texturizado con un sombreado parecido al de dibujos animados. La textura es la que da apariencia y color al modelo, mientras que el material se emplea, no como color, sino para determinar el nivel de sombreado. Por lo tanto, esto solo funciona cuando el modelo está texturizado, ya que si solo tiene material asociado, el color final no dependerá de su color, sino de los colores definidos en la tabla. Este efecto se muestra en la Figura 7.11.

El sombreado *highlight* es prácticamente igual que el toon, salvo que se realzan los colores que entrega la tabla toon sobre el color de la textura, aumentando el nivel de brillo de estas componentes, ganando peso sobre el color de la textura.

7.1.7. Anti-aliasing

En el proceso de rasterizado se determinan los fragmentos que corresponden a cada polígono de la escena 3D. Éstos son los elementos que más tarde corresponderán a los píxels de la pantalla. De esta forma, cada polígono pasa a estar formado por pequeños elementos cuadrilaterales (normalmente cuadrados). Por lo tanto, las líneas rectas de las aristas de los polígonos se forman a partir de una serie de cuadrados, por lo que acaban presentando una apariencia dentada.

Para disimular esta apariencia dentada se realiza el proceso de anti-aliasing, con el fin de

Figura 7.11: Sombreado toon aplicado a un modelo. Se puede apreciar la diferencia entre aplicar sombreado normal (izquierda) y sombreado toon (derecha) sobre un mismo modelo. En combinación con un contorno con el sombreado toon, se da una apariencia de dibujo animado más marcada.

difuminar los bordes de los objetos con el fondo y los objetos traseros. Éste consiste en aplicar una pequeña transparencia a los fragmentos de los bordes del objeto, de forma que la zona de unión entre el objeto y lo que tenga detrás se difumine.

En la Nintendo DS, este proceso es automático y no es configurable, pudiendo únicamente activarlo y desactivarlo. Para que tenga efecto, el identificador de los polígonos de los diferentes objetos y el del fondo han de ser diferentes, de lo contrario el chip gráfico no efectúa el proceso. Como es natural, es necesario configurar el fondo (color e identificador) para que este efecto se pueda aplicar.

El anti-aliasing no afecta a los polígonos translúcidos ni al interior de los polígonos (esto requeriría procesar la textura en la cual se genera la perturbación). Por otro lado, las aristas marcadas por el efecto de resalte de contornos se ven afectadas por el aliasing, a las que se les aplica un valor de transparencia, perdiendo contraste con el fondo.

7.1.8. Componentes de color del vértice simultáneas

Un punto especialmente importante en cuanto a la aplicación de parámetros de color a los vértices (y por lo tanto a los polígonos) es que se pueden aplicar varios atributos de color simultáneamente sobre un mismo vértice, puesto que el hardware los procesa por separado y los combina según el modo de mezcla especificado.

De esta forma, se pueden aplicar simultáneamente:

- **color directo**: este simplemente especifica el color del vértice.
- **color material**: este define las propiedades materiales del vértice. Este parámetro es el que reacciona a la iluminación mediante las normales, determinando el sombreado de la superficie que forman los vértices, así como un color correspondiente a cada componente del material (ambiente, difuso, especular y emisivo).
- **color de textura**: este se determina mediante las coordenadas de textura UV asignadas a cada vértice.

Así, se pueden aplicar simultáneamente diferentes propiedades de color a cada vértice,

pudiendo emplear esta capacidad para aplicar efectos más complejos gracias a la mezcla de todas las componentes. Por lo tanto, esto proporciona un elevado control sobre la apariencia de la malla.

7.2. Efectos posibles mediante software

En este apartado se exponen los efectos que se han conseguido implementar mediante algoritmos en el software, imposibles únicamente por hardware. De esta forma, utilizando los procesos que permite el hardware de la forma adecuada se pueden conseguir funcionalidades adicionales como la animación, tanto de geometría como de texturas, el renderizado multicapa (con el cual añadir efectos y detalles adicionales a los modelos), o efectos de *billboarding* (tanto esférico como cilíndrico).

7.2.1. Animación de geometría

La animación es un proceso que ha de realizarse en la etapa de aplicación, es decir, implementando algoritmos en el programa para aplicar dicha funcionalidad. Esto es cierto, al menos, hasta la aparición de los shaders (programas de la GPU, en particular los que actúan por vértice), permitiendo traspasar estos cálculos de la CPU a la GPU, aportando nuevas técnicas de animación, como la mezcla de varias animaciones, con un gasto computacional menor.

Por lo tanto, dado que la Nintendo DS no soporta shaders, es necesario realizar el procesado de animación en la aplicación de renderizado, empleando para ello las técnicas que permita el pipeline de su hardware gráfico.

En concreto se han podido implementar dos métodos de animación: la animación por transformación y la animación por muestreo de vértices, cuyo ámbito de aplicación viene determinado por el número de articulaciones que controlan una malla.

Animación por transformación

En este método, todos los vértices de una malla se animan siguiendo la transformación de la articulación que los controla. Esta transformación se muestrea para cada frame de animación. Así, mediante una sucesión de multiplicaciones matriciales, correspondientes a la posición y la rotación de la articulación en cada frame de animación, aplicadas a todos los vértices de la malla, se consigue el efecto de movimiento. En éste, la malla se mueve como un bloque rígido.

Como requisito lógico para que este método sea aplicable, la malla ha de estar afectada únicamente por una articulación, dado que en este proceso no se tienen en cuenta los pesos de las articulaciones asociados a los vértices, de forma que no se pueden aplicar las transformaciones de varias articulaciones.

Recordemos que la estructura de huesos que componen el esqueleto de un modelo está jerarquizada, de forma que la articulación hija hereda la transformación del padre. Por lo tanto, las transformaciones de las articulaciones suelen exportarse relativas a la del padre para cada clave de animación (MilkShape 3D así lo hace). Sin embargo, en este caso, para

aligerar la carga computacional del procesador de la consola, el conversor EDL calcula las transformaciones absolutas de todas las articulaciones en cada frame de animación, teniendo en cuenta las transformaciones de las articulaciones anteriores. De esta forma, en la aplicación de renderizado, en cada frame de animación, tan solo hay que aplicarle a cada malla su transformación correspondiente, independientemente de la del padre.

Como funcionalidad adicional la biblioteca permite al usuario leer la transformación de las articulaciones, que puede ser aplicada a otros objetos para situarlos de forma coherente con el modelo. Esta transformación puede servir para equipar un personaje con un objeto en su mano, colocar elementos móviles en un escenario, etc.

Animación por muestreo de vértices

En la técnica de animación denominada *skinning animation*, los vértices de una malla pueden estar afectados por varias articulaciones, determinando la cantidad de control que éstas ejercen sobre ellos, a través de pesos asociados a cada uno.

El método más sencillo para aplicar ésta forma de animación es enviar la posición actualizada de los vértices en cada frame da animación. De esta forma, en cada frame el modelo adquiere una postura que va evolucionando a través de la línea temporal de la animación, generando así el movimiento.

Por lo tanto, el conversor EDL almacena la posición de los vértices afectados por este método de animación para cada frame. De esta forma, en cada ciclo NDS (60 fps) se mandan los vértices con sus coordenadas correspondientes al frame de animación actual.

Actualización del frame de animación

Como ya se ha comentado, la Nintendo DS realiza 60 ciclos por segundo. Por lo tanto, es necesario adaptar la frecuencia de la animación a la de trabajo del procesador de la consola.

Para ello, el conversor EDL proporciona un valor referente al periodo de animación, que se traduce como el número de ciclos NDS que dura un frame de animación. De esta forma, cuando este número de ciclos haya transcurrido se actualiza el frame de animación.

Dependiendo del modo de reproducción de la animación se actualiza en un sentido o en otro. Se pueden considerar tres modos:

- **forward**: se actualiza al siguiente frame de animación.
- **backward**: se actualiza al frame de animación anterior.
- **ping-pong**: se recorre el clip de animación en ambos sentidos consecutivamente.

7.2.2. Animación de texturas

Debido a la limitación que presenta la consola en cuanto al número de polígonos que se pueden pintar, es interesante poder realizar efectos de movimiento que no requieran desplazamientos de geometría. Para ello se puede emplear la animación de texturas.

Considérense dos formas de animación de texturas:

- **Animación progresiva**: se transforman las coordenadas de textura mediante multiplicaciones matriciales, consiguiendo desplazamientos, rotaciones y escalados que

producen movimientos fluidos. Para ello, en cada ciclo NDS se ha de variar mínimamente la transformación y así producir este movimiento continuo.

- **Animación por subimágenes**: la imagen de textura se divide en varias subimágenes cuya sucesión corresponde a una animación (tal y como en la animación tradicional). En la fase de modelado se limitan las coordenadas de textura a una de las subimágenes. De esta forma, en la aplicación de renderizado, para generar la sucesión de imágenes se desplazan las coordenadas de textura de tal forma que coincidan con la siguientes subimágen. Para ello se tienen en cuenta el número de subimágenes en horizontal y en vertical, y las dimensiones de la textura en píxels. De esta forma se puede calcular el número de píxels que se deben desplazar las coordenadas de textura mediante las expresiones [ancho/subimágenes en horizontal] y [alto/subimágenes en vertical]. Así, desplazando la textura en cada actualización del frame de animación se consigue un movimiento animado, sin necesidad de realizar cálculos de geometría.

7.2.3. Renderizado multicapa

La superposición de capas de pintado se emplea para añadir detalle a la apariencia del modelo, el cual no se puede conseguir únicamente con el material y una textura.

El hecho de separar las capas, además de por la imposibilidad de aplicar varios efectos en una sola pasada, es interesante para independizar los efectos que aporta cada una. De esta forma, se pueden activar y desactivar independientemente. Puede ser interesante, por ejemplo, para añadir una capa de suciedad al modelo según pasa el tiempo, sin la necesidad de aplicar esta apariencia de suciedad a la textura principal del material.

La renderización de varias capas, en un sistema cerrado como el pipeline gráfico de la Nintendo DS, se puede lograr enviando los mismos vértices que componen el modelo, afectados por las propiedades materiales correspondientes a cada capa adicional. Como es lógico, únicamente hay que enviar los vértices afectados por estos efectos. Esto es porque de esta forma se disminuye el número de polígonos pintados, ahorrando dicha memoria, y porque si se aplica transparencia de mezcla 0, se pinta el polígono correspondiente en modo alambre, produciendo un efecto indeseado.

Para que estos mismos vértices se pinten superpuestos sobre los anteriormente enviados al hardware, es necesario activar el bit de test de profundidad del parámetro de atributos del polígono (Depth Test) en cada capa adicional. Esto permite pintar únicamente los píxels con la misma profundidad que los ya presentes en el *depth buffer*. De esta forma se pintan los nuevos polígonos superpuestos a los anteriores, con diferente apariencia.

Realizar únicamente esta operación no bastaría, ya que las capas pintadas en última instancia taparían las capas inferiores. Para evitar este efecto indeseado, se la aplica a cada capa adicional un valor de transparencia de mezcla. Así, las capas inferiores se visualizan en la medida en que la transparencia lo permita.

El orden de pintado de las capas determina la aplicación de un efecto sobre las demás capas. Las capas superiores aplican su efecto sobre las inferiores y no a la inversa. Por lo tanto, es necesario determinar correctamente el orden de pintado para que el efecto aplicado sea el deseado. Un ejemplo de esto es un objeto metálico afectado por el efecto de la reflexión esférica al que se le añade una capa de suciedad. No interesa que ésta

última se vea afectada por la reflexión, por lo que deberá pintarse primero la capa de reflexión, seguida de la capa de segunda textura que representa la suciedad.

En la herramienta de renderizado msNDS únicamente se han implementado dos capas adicionales: la de segunda textura (Figura 7.12) y la de reflexión (Figura 7.13). Sin embargo, se pueden pintar tantas capas se quiera, atendiendo al consumo que éstas conllevan, dado que en cada capa se pinta nuevamente el modelo completo (al menos todas las mallas afectadas). Por ello es necesario limitar los objetos a los cuales se les aplican estos efectos y el número de efectos aplicados, con el fin de no consumir todos los polígonos representables en el pintado de unos pocos objetos y no dejar margen para pintar el resto de modelos.

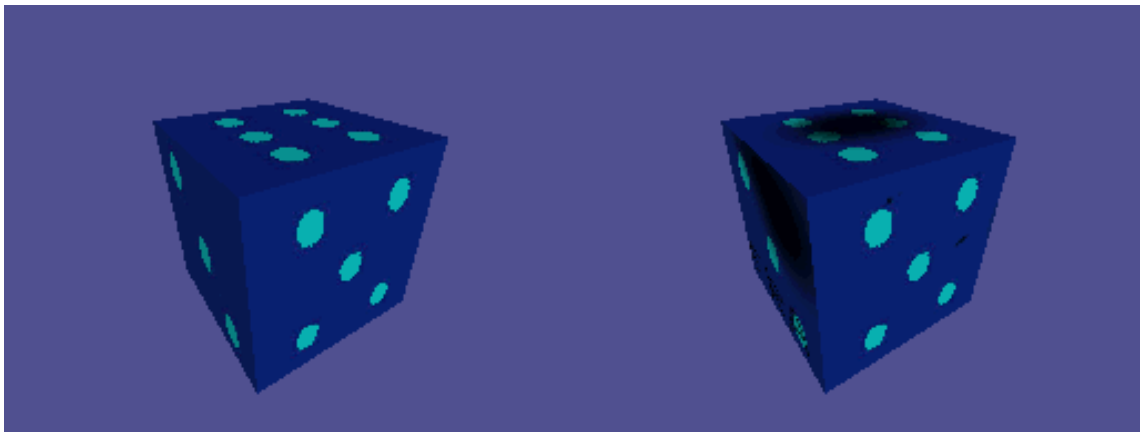

Figura 7.12: Aplicación de una segunda textura sobre la textura principal del dado. Ésta puede simular suciedad o cualquier otra apariencia extra.

7.2.4. Billboarding

El efecto de *billboarding* consiste en que el objeto al que se le aplica actualiza su rotación con el fin de encarar siempre a la cámara. De esta forma, el usuario siempre ve la parte frontal del objeto (de ahí su nombre, ya que la palabra inglesa *billboard* se traduce como los carteles publicitarios de las carreteras norteamericanas, que parece que siempre apuntan hacia el coche).

Este efecto se suele emplear para representar objetos complejos (como los objetos que rellenan la escena) mediante una imagen de éstos, sin necesidad de emplear geometría. Así, un árbol se puede representar con una imagen de éste, con transparencia en aquellas zonas donde no haya dibujo del árbol, aplicada como textura a un plano. De esta forma, si el árbol tiene simetría cilíndrica y gira alrededor de la zona que representa el tronco, al estar siempre de frente a la cámara, nunca se tiene consciencia de que se trata de un plano y se ve siempre la imagen del árbol.

Además, se puede combinar con la animación de texturas por subimágenes para representar efectos como el fuego. En este caso, la animación crearía el movimiento de la llama y el billboard haría que siempre se vea de frente, con lo que en cualquier posición se apreciaría la llama crepitando.

El *billboard* se aplica a la transformación de una articulación, afectando a todas las mallas controladas por ésta. Existen dos tipos de *billboard*, que son:

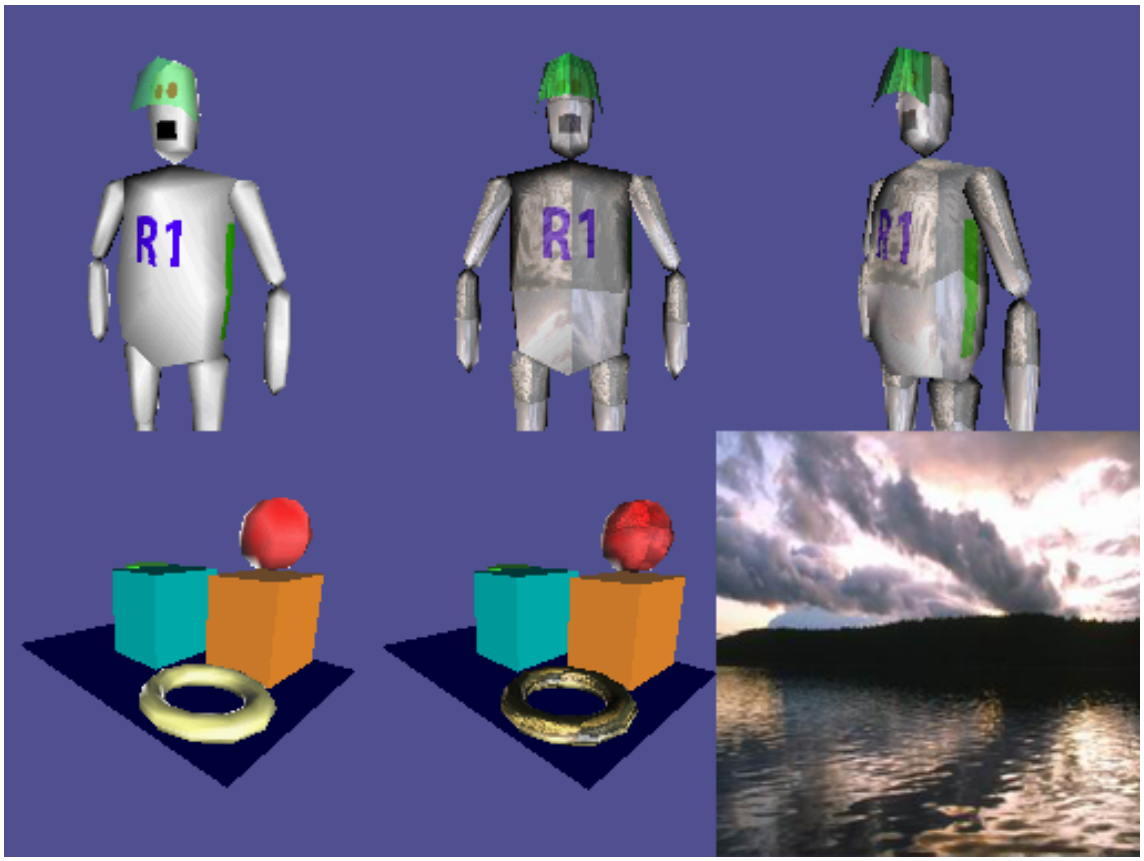

Figura 7.13: Capa de reflexión esférica aplicada sobre un modelo para darle un aspecto metalizado. El efecto pretendido es la aplicación de la textura (abajo a la derecha) sobre el modelo y que ésta siempre quede de frente a la cámara y se deforme según su geometría. En la librería msNDS se ha conseguido este efecto; sin embargo no se ha conseguido que la textura quede bien escalada y centrada sobre el objeto, por lo que se ven las esquinas de la imagen en el centro del modelo.

- ***billboard* cilíndrico**: el objeto modifica su rotación en el eje Y para encarar a la cámara. La rotación horizontal queda intacta, por lo que se aprecia el efecto si la cámara se posiciona sobre el objeto. Es útil para objetos con simetría cilíndrica como árboles, farolas, postes, etc.
- ***billb*oard esférico**: el objeto modifica su rotación tanto en el eje Y como en el eje X y el eje Z, de forma que siempre queda de frente a la cámara, sin importar si ésta se coloca encima de dicho objeto. Es útil para objetos con simetría esférica como explosiones, bolas, nubes, etc.

Una técnica sencilla de realizar este efecto, que no requiere cálculos complejos (ideal para un sistema más limitado como es la Nintendo DS) es modificar la matriz de modelo (*modelview*) en el momento de pintar el objeto. Dependiendo del tipo de *billboard* se realiza una operación determinada sobre esta matriz. En el caso del *billboard* esférico, se elimina la rotación de la matriz de modelo, quedando la articulación de frente a la cámara (siempre que la posición inicial del modelo exportado sea ésta). Para el *billboard* cilíndrico se elimina la rotación de la matriz de modelo, manteniendo únicamente la rotación cenital (eje y).

Este proceso viene explicado en detalle en el capítulo dedicado a la librería de renderizado

msNDS, en el apartado referente al funcionamiento interno de la aplicación.

7.3. Limitaciones

El motor gráfico de la Nintendo DS, como cualquier sistema, presenta sus virtudes y sus defectos. Se trata de un procesador gráfico cerrado, por lo que no se tiene total libertad a la hora de realizar las operaciones gráficas, sino que únicamente se pueden realizar ciertas configuraciones del estado del hardware. Esto supone limitaciones en cuanto a lo que se puede realizar y la manera de hacerlo, provocando en ciertas ocasiones muchas dificultades e incluso imposibilidades a la hora de implementar algún efecto.

Además se trata de un chip gráfico relativamente antiguo, lo que añade las limitaciones propias del envejecimiento tecnológico, y más teniendo en cuenta que la Nintendo DS tampoco era una máquina demasiado avanzada en la fecha de su lanzamiento. Esto se refleja principalmente en la memoria disponible, muy escasa, y en el ya comentado pipeline cerrado que presenta.

Por lo tanto, en este apartado se comentarán estas limitaciones y los obstáculos que se han encontrado en el desarrollo de este proyecto.

7.3.1. Memoria

Pese a que ya se ha comentado la limitación de la memoria, tanto principal como de vídeo, éste es el mayor hándicap del procesador gráfico de la consola, y por lo tanto merece un apartado adicional en este capítulo dedicado a las capacidades gráficas de la Nintendo DS.

Memoria principal

La memoria principal se emplea para cargar el programa y las variables creadas durante la ejecución de éste. Además, es también donde se cargan los recursos importados del exterior, como modelos, imágenes o archivos de audio.

La Nintendo DS dispone de 4096 kB de memoria RAM principal. Teniendo en cuenta que un modelo (en formato edl) con sus texturas puede ocupar entre 20 kB y 150 kB (dependiendo de la densidad de vértices que tenga), este valor puede resultar escaso, en el caso de que se carguen el programa, varios modelos con sus texturas y archivos de audio.

Sin embargo, existe la posibilidad almacenar los recursos como archivos en la memoria del cartucho o tarjeta desde donde se ejecuta el programa y cada vez que se vaya a usar alguno de éstos cargar los datos de fichero. Se emplea menos memoria RAM, pero la carga de los datos de fichero es lenta, por lo que no se debe abusar ni de un método ni de otro. Lo lógico es por ejemplo cargar en memoria principal todos los recursos de una escena al principio de ésta (lo que probablemente requiere algún tiempo de carga) y eliminarlos cuando se pase a la escena siguiente. De esta forma se pueden gestionar los recursos para no desperdiciar la escasa memoria.

Memoria de video

La memoria de video es probablemente la más escasa y la más compleja de gestionar, debido a que está dividida en nueve bancos de memoria virtuales. Ésta consta de 656 kB para almacenar texturas, paletas asociadas, sprites y fondos, que se cargan en los diferentes bancos destinados a tales elementos.

Por una parte, la escasez de memoria repercute en el número de imágenes que se pueden cargar simultáneamente. Esto implica que puede darse el caso de que se cargue un modelo pero no haya espacio en memoria para sus texturas, provocando que éstas no se representen. Por lo tanto es vital tanto crear texturas ajustadas en tamaño (cumpliendo con las exigencias artísticas), como gestionar adecuadamente su carga en memoria de video.

Un aspecto importante en este sentido es la reutilización de texturas siempre que sea posible, ya que una vez esté una textura en memoria se puede emplear cuantas veces sea necesario, aunque sea para modelos diferentes.

Memoria 3D (vértices y polígonos)

La última limitación de memoria correspondiente a los gráficos tridimensionales en la Nintendo DS es la memoria 3D. Ésta es la correspondiente a los vértices y polígonos enviados al hardware, limitada en 144 kB para vértices y 104 kB para polígonos. Esto repercute en la cantidad de geometría que se puede renderizar en pantalla.

Indirectamente, esta limitación repercute en el tamaño (en cuanto a densidad de polígonos) de los modelos. Si un modelo tiene demasiados vértices, no quedará memoria para los demás, por lo que el procesador gráfico no podrá pintarlos. Este es, por lo tanto, otro aspecto que gestionar: es necesario minimizar el número de polígonos de los modelos atendiendo a las especificaciones artísticas.

Por otra parte, dado que las pasadas de render hacen uso de geometría adicional, la limitación de memoria 3D también repercute en las capas de render que se pueden aplicar por modelo. Si se emplean demasiados vértices para pintar un modelo con varias capas de render, no se dispondrá de memoria suficiente para pintar los modelos restantes (se puede dar el caso de emplear toda la memoria en un solo modelo poligonalmente denso con dos pasadas de render).

Con el fin de ahorrar geometría sin renunciar a crear escenarios vistosos y detallados, es conveniente recurrir a técnicas como el *billboard* y la animación por subimágenes. Con ellas, se crea la apariencia de objetos y efectos con la geometría simple de un plano, ahorrando así en memoria 3D utilizable en objetos importantes y significativos.

7.3.2. Arquitectura cerrada del pipeline gráfico

Al igual que el resto de procesadores gráficos anteriores a la irrupción de los shaders en le escena de gráficos por ordenador, el motor 3D de la Nintendo DS se basa en un una cadena de operaciones fija, no programable por el desarrollador. Ésta tan solo puede ser configurable, y en el caso de esta consola las opciones de configuración del hardware son escasas.

Por esta razón, los efectos gráficos que se pueden implementar están muy condicionados a

la arquitectura del pipeline, por lo que, para realizar algunos de éstos más complejos hay que realizarlos por software. Esto consume recursos y ciclos de computación, aumentando el flujo de operaciones y ralentizando el proceso, por lo que no siempre será conveniente realizarlos.

Por lo tanto, este tipo de arquitectura, que por otra parte simplifica enormemente el proceso de desarrollo, ya que son pocas las operaciones posibles, no permite la libertad de actuación que aportan los sistemas actuales basados en procesadores de shaders, en los que todas las operaciones son programables por el desarrollador, permitiendo cualquier efecto posible.

7.3.3. Concatenación de capas

El desarrollo de la herramienta de renderizado msNDS ha implicado la implementación de una serie de funcionalidades establecidas en los objetivos iniciales. Una de estas funcionalidades es el renderizado multi-capa, en el que se realizan varias pasadas de render, aplicando unos efectos o apariencias adicionales al modelo.

En concreto se han implementado dos pasadas de render adicionales, tal y como se indica en el capítulo referente a la librería msNDS. Una de ellas es la aplicación de una segunda textura, con la cual añadir más detalle de forma controlada por el usuario. La segunda es una capa de reflexión esférica, que simula le reflexión del entorno producida sobre objetos metálicos para dar esta apariencia a los modelos. El orden en el que estas capas se pintan se puede determinar mediante directivas al conversor EDL, que añade este parámetro a los datos del modelo.

Sin embargo, se ha encontrado un funcionamiento inesperado a la hora de pintar las capas de segunda textura y reflexión en este orden. En este caso, la capa de segunda textura no se pinta, quedando visible únicamente la de reflexión. Este es un hecho desconcertante, ya que en el orden inverso no hay ningún problema. Además, otro indicativo de que se trata de un problema con el hardware de la consola es que en los emuladores esto no sucede.

Es posible que se trate de un fallo en la aplicación, que se manifiesta de diferentes formas en una plataforma u otra. Sin embargo, también es posible que, dado el proceso interno realizado por el procesador gráfico esta operación ocasione algún conflicto, impidiendo tal efecto.

De cualquier forma, se trata de un error presente en la aplicación, por lo que será objeto de revisión en futuras versiones de ésta.

Capítulo 8

Conclusiones y trabajo futuro

En este capítulo final se recopilan las conclusiones extraídas a lo largo del desarrollo del proyecto. Con esto se determinan los puntos destacados de la aportación de éste al desarrollo 3D en Nintendo DS y se analizan la consecución de los objetivos propuestos.

Además, se expondrán unas posibles líneas de trabajo futuro para ampliar el tema tratado en este proyecto y el programa implementado en éste.

8.1. Cumplimiento de objetivos

Al inicio del proyecto se determinaron una serie de objetivos que cumplir referentes a las posibilidades del motor 3D de la Nintendo DS y el desarrollo de una herramienta de programación que aproveche estas capacidades y facilite el desarrollo 3D en dicha consola.

Por lo tanto, al término del proyecto, se analiza la consecución de dichos objetivos, exponiendo a la vez la contribución de este trabajo al tema de los gráficos 3D en la Nintendo DS.

8.1.1. Implementación de la librería de renderizado 3D msNDS

Por un lado se ha implementado una biblioteca de renderizado en C, que facilita el desarrollo de aplicaciones tridimensionales en la Nintendo DS, accediendo a las capacidades 3D de ésta. Ésta ha sido desarrollada sobre la parte gráfica de la API Libnds, creada bajo la cadena de herramientas DevkitPro para facilitar el desarrollo en Nintendo DS.

Funcionalidades

La biblioteca msNDS consta de una serie de funcionalidades con las cuales aplicar procesos o efectos relacionados con el motor 3D de la Nintendo DS y así renderizar modelos tridimensionales en pantalla con apariencias más o menos detalladas y complejas (dentro de las posibilidades del hardware).

Así, la librería permite:

- **cargar modelos 3D en formato EDL** (Extended Display List), específicamente diseñado para este propósito e implementado en paralelo a este proyecto, que

incluyen geometría (coordenadas de vértices), normales, coordenadas de textura, materiales y color por vértice de cada malla del modelo e información de animación.

- crear varias instancias de un mismo modelo independientes entre sí en cuanto a estados de animación
- aplicar **texturas** a estos modelos, en cualquiera de los formatos con los que trabaja la consola (menos el formato comprimido): RGB (color directo), RGB4, RGB16, RGB256, A5I3 y A3I5.
- aplicar **color por vértice**, permitiendo añadir un parámetro de control de apariencia adicional
- realizar varias **pasadas de render**, en cada cual aplicar una capa de apariencia adicional, estando implementadas las capas de **segunda textura** y de **reflexión esférica** (esta última no funciona correctamente, por lo que sería un punto a tratar en futuras revisiones; no obstante sirve para demostrar que se puede realizar renderizado multi-pasada con varias capas simultáneas)
- reproducir **animaciones**. Se reproducen independientemente tanto las animaciones de cada instancia de un modelo como las animaciones propias de cada esqueleto de un modelo. Se pueden reproducir las animaciones en sentido directo, inverso o en modo ping-pong, en el que se recorre la animación en ambos sentidos consecutivamente. Los dos posibles modos de animación son:
  - **animación por transformación**: se transforman, mediante cálculos matriciales, las mallas del modelo que permitan este modo. Éste solo es posible si todos los vértices de la malla están controlados por una única articulación.
  - **animación por muestreo de vértices**: se almacenan los vértices afectados por este modo en cada muestra de animación, capturando la pose del modelo en cada frame. Este modo es necesario cuando una malla está animada por la técnica de *skinning animation*, en la cual cada vértice puede estar controlado por articulaciones diferentes.
- **animar texturas por subimágenes**: creando una textura con varias imágenes que constituyen una animación, se puede simular movimiento sin animar geometría.
- aplicar restricciones **billboard**: se modifica la rotación de la articulación afectada para que apunte siempre en dirección a la cámara, de forma que siempre se visualice su parte frontal. Dos métodos de billboard:
  - **esférico**: se modifica la rotación del objeto alrededor de un punto (la articulación), de forma que éste describe un movimiento rotatorio esférico para posicionarse frente a la cámara, por lo que siempre se visualiza su parte frontal.
  - **cilíndrico**: únicamente se modifica la rotación del objeto en el eje vertical, de forma que el objeto sigue la ubicación de la cámara mediante un movimiento rotatorio cilíndrico, pero si ésta se posiciona sobre el objeto se puede llegar a visualizar su parte superior.
- **gestionar recursos**: se pone a disposición un sistema de gestión de recursos con el cual cargarlos y extraerlos de listas para que sea sencilla su manipulación. En la versión actual del programa únicamente están disponibles recursos de tipo textura,

por lo que ampliar a más tipos de recursos podría ser una línea de desarrollo futuro.

Operatividad

Con el fin de aportar un nivel de control robusto sobre el sistema de renderizado, todas las funcionalidades implementadas en la biblioteca msNDS pueden ser activadas o desactivadas, siempre que éstas estén disponibles en el modelo. De esta forma, el usuario de esta herramienta tiene varias opciones de renderizado a su disposición y total control sobre su representación.

Por otra parte, la librería ha sido diseñada de forma modular, por lo que es utilizable externamente en cualquier proyecto que lo requiera.

Uno de los requisitos iniciales en el desarrollo de msNDS es que aporte facilidades a sus usuarios para crear aplicaciones 3D, por lo tanto se ha diseñado de tal forma que su utilización sea sencilla e intuitiva. Esto se basa en la implementación de funciones de acceso sencillas de emplear para que el usuario no requiera más que un mínimo aprendizaje para sacar partido de todas las capacidades del sistema de renderizado de msNDS.

Por lo tanto, se ha implementado una herramienta cuyo uso es sencillo, que aporta funcionalidades que se ajustan a las posibilidades técnicas de la Nintendo DS y que ofrece un nivel de control casi completo, facilitando así el desarrollo de juegos 3D, sin tener que preocuparse del complejo proceso que conlleva el renderizado de gráficos.

8.1.2. Capacidades de la Nintendo DS

Mediante el trabajo de investigación y el desarrollo efectuados sobre la Nintendo DS se han ido determinando las posibilidades que ofrece su procesador gráfico y las operaciones que éste puede efectuar.

En la medida que el chip gráfico de la Nintendo DS sigue una cadena de procesamiento cerrada, el sistema de renderizado se puede controlar externamente en cierta medida. Por un lado, se pueden enviar los vértices y sus atributos (normales, coordenadas de textura, materiales/color), definiendo el tipo de primitiva que forman. Además, es posible establecer los estados del hardware que configuran los parámetros del renderizado, accediendo para ello a los registros del procesador correspondientes.

De esta forma, únicamente se pueden configurar ciertos parámetros permitidos por el procesador gráfico, lo que limita en gran medida las posibilidades referentes al pintado de objetos 3D.

Sin embargo, ofrece ciertos efectos gestionados internamente (que se pueden activar/desactivar) con los cuales se consiguen resultados vistosos de forma sencilla. Éstos son:

- **resalte de contornos**, con el cual se puede dar la apariencia de dibujos mediante una línea dibuja la silueta de los objetos, o con el que resaltar objetos seleccionados
- **efecto niebla**, que difumina los objetos lejanos, mezclando con un color de fondo y de esta forma remarcando la sensación de profundidad
- **sombreado toon**, con el cual se aplica una apariencia de dibujos animados simple pero vistosa

- **anti-aliasing**, que difumina los contornos de los objetos con el fin disimular la pixelación en estas zonas límite

Además de estos efectos característicos y propios de la consola, merecen especial mención dos características referentes al envío de vértices al hardware. La primera es la posibilidad de aplicar simultáneamente varios parámetros de aspecto por vértice, que son: color directo (color por vértice), color material (el que reacciona a la iluminación) y el color de textura (determinado mediante las coordenadas UV). Estos tres parámetros (en caso de que se especifiquen) se mezclan según el modo de blending especificado al hardware. Esto posibilita la aplicación de efectos muy vistosos, que aportan detalle al conjunto del modelo.

La segunda característica destacable referente al envío de vértices es la posibilidad de enviar varias veces un mismo vértice (es decir, la misma geometría), operación que permite realizar varias pasadas de render. Con esto, una vez más, se aplica más detalle y complejidad al aspecto final del modelo renderizado.

La Nintendo DS, si bien trabaja con un procesador limitado, tanto en memoria (gran hándicap de la consola) como en opciones de renderizado, dispone de unas capacidades gráficas interesantes, sobre todo por el hecho de la relativa sencillez a la hora de ponerlas en marcha (una vez se conocen los entresijos del hardware) y los efectos que se pueden sacar exprimiendo un poco su motor 3D.

Por lo tanto, y en definitiva, se trata de una plataforma muy interesante sobre la que desarrollar, tanto por sus propias cualidades como por la extensa escena *homebrew* que le da soporte. En este último caso, se requiere un poco más de profundidad en el aspecto de los gráficos 3D, por lo que este proyecto se presenta como un pequeño paso más hacia delante en esta materia.

8.2. Líneas de trabajo futuro

En este apartado se presentan unas vías de trabajo futuro, concebidas como posibles mejoras y ampliaciones, que por tiempo y recursos no ha sido posible implementar en la versión actual de la librería msNDS.

8.2.1. Mejora y ampliación del gestor de recursos

En estos momentos, el gestor de recursos, si bien realiza su función correctamente, que es la de enlazar las texturas de los modelos cargados con el sistema de renderizado, ésta se queda algo corta, ya que tan solo se puede emplear para el almacenaje y manipulación de texturas, quedando otros posibles recursos, como clips de sonido o los propios modelos, excluidos de este sistema.

Actualmente el sistema solo funciona con recursos residentes en memoria, siendo una mejora importante poder cargar tanto texturas como sus paletas y los modelos desde fichero, o bien desde la imagen del programa (Nitro filesystem) o desde la memoria flash (FAT filesystem).

Por lo tanto, se podría mejorar el sistema de gestión de recursos existente, haciéndolo más eficiente, y ampliando los tipos de recurso manejables por éste.

8.2.2. Corrección del efecto de reflexión esférica

El efecto de reflexión esférica implementado en el sistema de render de msNDS, si bien en algunas ocasiones parece funcionar de manera similar a la esperada, no funciona perfectamente en todos los casos, emitiendo resultados indeseados y, por lo tanto, no utilizables en proyectos externos.

Aunque cumple una de sus funciones, que es la de comprobar la viabilidad de realizar varias pasadas de renderizado, es una funcionalidad interesante, por lo que debería ser mejorada en futuras versiones del programa.

8.2.3. Animación con interpolación

Como se ha comentado, el sistema de animación incorpora dos técnicas para mover la geometría frame a frame, una por muestreo de las transformaciones de las mallas, y la otra por muestreo de vértices. Aunque se trata de métodos bien distintos, ambos comparten la forma en la que se avanza sobre el clip de animación.

El avance se realiza mediante el periodo de animación, almacenado en el propio modelo, que indica el tiempo transcurrido, en ciclos NDS (60 fps), entre frame y frame de animación. De esta forma se determinan los ciclos que deben transcurrir entre muestra y muestra, actualizando así los frames de animación.

Sin embargo, este sistema podría ser mejorado realizando una interpolación de las transformaciones en cada ciclo, independientemente de si se está o no sobre un frame de animación. De esta forma, el movimiento, independientemente de la velocidad o el periodo de la animación, queda más fluido, dando una apariencia de continuidad.

8.2.4. Mejoras de velocidad

Dos aspectos que mejorarían la velocidad del rendering serían el uso de *triangle strips* y el uso de *display lists*.

Toda la geometría se representa con la primitiva Triangles que manda los tres vértices de cada triángulo al hardware gráfico. Esto implica que los cálculos realizados por vértice en cuanto a transformación e iluminación pueden repetirse innecesariamente. En efecto, cuando varios triángulos comparten un mismo vértice con parámetros iguales (normal, coordenadas de textura, color) podría aprovecharse esta circunstancia para ahorrar cálculos si ambos vértices forman parte de una primitiva Triangle Strip (ristra de triángulo). Existen algoritmos para transformar una malla de triángulos independientes en un conjunto de tiras o ristras de triángulos que en el mejor de los casos dividirían por dos las necesidades de cálculo por vértice del hardware gráfico.

El hardware gráfico de la consola puede recibir además de comandos independientes como envía msNDS ahora una secuencia de comandos. En efecto, los comandos de representación 3D se pueden escribir en una sintaxis comprimida que se denomina *Display List*. La ejecución de una de estas listas de comandos es más rápida que una secuencia de comandos independientes. Además la CPU puede ejecutar otro código diferente mientras la GPU procesa la lista, mientras que en la implementación actual la CPU espera con cada comando.

Una mejora muy interesante sería estructurar cada malla en forma de una o varias de estas listas, de forma que su representación sería más rápida. Además si estas listas se combinan con la estructuración en Triangle Strips el espacio de almacenamiento para las listas se reduciría y se tendrían el doble beneficio de velocidad y ahorro de memoria.

8.2.5. Nintendo 3DS

La nueva versión de la consola de Nintendo la 3DS, además de permitir gráficos autoestereoscópicos cuenta con una GPU mucho más potente. Un tema de investigación futuro muy interesante sería portar esta biblioteca de funciones a dicha plataforma además de explorar sus nuevas posibilidades. Desgraciadamente la comunidad *homebrew* no cuenta con herramientas que permitan utilizar la consola en modo 3DS, con lo que este desarrollo tendrá que esperar.

Apéndices

Apéndice A

Especificación del formato Extended Display List

A.1. Estructura de los datos en el formato EDL

En esta sección se especifica el formato Extended Display List, uno diseñado para albergar los datos en los formatos que maneja la Nintendo DS. Así, se detalla la estructura de los datos almacenados en un fichero EDL durante el proceso de conversión (mediante el conversor EDL). Ésta está reflejada en la Tabla A.1.

En esta tabla, se estructuran los datos en bloques de 32 bits, correspondientes a cada fila en las que se exponen dichos datos. Esta estructuración corresponde a la forma en la que se escriben (y se han de leer) los datos en el fichero. En cada caso, los bloques se dividen en secciones de menor tamaño (como mínimo de 1 byte), dependiendo del tamaño de los datos que se alberguen en cada uno.

En algunos casos, hay secciones dentro de un bloque de 32 bits que quedan inutilizadas, dado el diseño del formato. Éstas vienen representadas en la tabla con una barra horizontal.

La estructura que se presenta a continuación corresponde a la versión 3 del formato EDL, por ser la más completa.

Tabla A.1: Especificación de la versión 3 del formato EDL.

<table>
<tr><td>33h</td><td>4Ch</td><td>44h</td><td>45h</td><td>EDL3 caracteres para indicar comienzo del tipo de fichero</td></tr>
<tr><td>NJ</td><td>NS</td><td>NG</td><td>NM</td><td>Componentes del modelo</td></tr>
<tr><td colspan="4"></td><td>NM número de materiales 8 bits
NG número de grupos/mallas 8 bits
NS número de esqueletos 8 bits, no puede valer 255
NJ número de articulaciones 8 bits, no puede valer 255</td></tr>
<tr><td colspan="3">—</td><td>flags</td><td>Opciones de modelo</td></tr>
<tr><td colspan="4"></td><td>R bit 0: indica si hay materiales reflectivos para capa de reflexión esférica
T bit 1: indica si hay algún material con segunda textura animada
O bit 2: indica el orden de las capas de reflexión/segunda textura, a 1 2ª textura/reflexión, 0 viceversa</td></tr>
<tr><td colspan="5">Para cada esqueleto de los NS esqueletos en el modelo</td></tr>
<tr><td></td><td>Period</td><td>Samples</td><td colspan="2">Datos de muestreo de la animación en el esqueleto</td></tr>
</table>

				Samples 16 bits: número de muestras **Period** 16 bits: periodo de muestreo en formato 8.8 medido en número de frames (NDS 60 fps)
	—		NC	**NC** 8 bits: número de costumbres de animación
				Si NC = 0, es que el esqueleto no está animado Si NC = 0, es que no viene lo siguiente, la muestra final es Samples - 1
	Ends(1)	Ends(0)		Muestra final de cada costumbre (16 bits), la primera va de 0 a Ends(0) la segunda de Ends(0) a Ends(1) - 1, etc.
	...	Ends(2)		
Para cada articulación de las **NJ articulaciones** en el modelo				
	—	flags	SK	
				SK 8 bits: índice del esqueleto al que pertenece la articulación **flags**: **BB** bits 0-1: 0 nada, 1 billboard cilíndrico, 2 billboard esférico **T** bit 2: hay datos de traslación **R** bit 3: hay datos de rotación
	TX			**Traslación en reposo**: tres datos en formato **f32** Una malla que depende completamente de esta articulación necesita esta traslación en reposo
	TY			
	TZ			
	Si **T** para cada una de las **Samples** del esqueleto **SK**			
		TX		**Traslación** para la muestra n, 3 datos en formato **f32**
		TY		
		TZ		
	Si **R** para cada una de las **Samples** del esqueleto **SK**			
		XA		**Rotación** para la muestra n, 4 datos en formato **f32** **Eje** y **ángulo en grados**, en formato **f32**
		YA		
		ZA		
		angle		
Para cada malla de las **NG mallas** en el modelo				
	—	joint	ske	flags	Opciones de la malla
					S bit 0: objeto **sólido**, hacer *backface culling* **N** bit 1: incluye **normales** **T** bit 2: incluye **coordenadas de textura** **C** bit 3: incluye **color por vértice** cpv **V** bit 4: a 1 vértices en formato **V16**, sino **V10** **WS** bit 5: **coordenada horizontal** de textura necesita **repetición** **WT** bit 6: **coordenada vertical** de textura necesita **repetición** **A** bit 7: la malla está **animada** **ske** 8 bits: **esqueleto** que la anima o 255 sin animación **joint** 8 bits: **articulación** que anima esta malla, toda la malla depende de ella Si es 255 la malla no puede animarse por transformación
	—	MI	NN	NV	Tamaño en componentes de la malla e índice a tabla de materiales
					NV 10 bits: número de vértices **NM** 10 bits: número de normales (útil solo si bit N activado) **MI** 8 bits: índice del material de la lista que viene después
	Para cada vértice sus **Coordenadas**. Dos opciones:				
	-	Z	Y	X	Formato **V10**, 10 bits por coordenada, formato 4.6. Si bit **V** es 0 [1]

[1] En este caso los bloques son de 10 bits, correspondientes a cada coordenada, y únicamente se dejan inutilizados 2 bits, y no un bloque completo de 1 byte.

	Y		X		O formato **V16**, 16 bits por coordenada, formato 4.12. Si bit **V** es 1
	—		Z		
	Para cada vértice, sólo si bit **T** activado				
	T		S		**Coordenadas de textura**, 16 bits cada una
					Formato independiente del tamaño de imagen
	Para cada vértice, sólo si bit **C** activado				
	Color n+1		Color n		**Color por vértice**, 5 bits por componente. 2 colores en 32 bits
					Si hay número impar, el último se rellena con ceros
	Para cada normal, sólo si bit **N** activado				
					Normal 10 bits por componente (V10), en formato 1.9
					Número de **triángulos** (sobra sitio)
	Para cada triángulo				
	—				**Índice** a cada **vértice**, 10 bits (máximo 1023 vértices)
	Para cada triángulo, sólo si bit N activado				
	—				**Índice** a cada **normal**, 10 bits (máximo 1023 normales)
	Muestreo de vértices y normales				
	Sólo si: **A=1** y **joint=255**, es una malla animada pero no animable únicamente por la transformación de **joint**. Muestras de todos los vértices: $NV \times Samples$ (de **ske**). Primero todos los vértices en orden de la primera muestra, y así de muestra en muestra.				
	-	Z	Y	X	Formato **V10**, 10 bits por coordenada, formato 4.6. Si bit **V** es 0 [1]
	Y		X		O formato **V16**, 16 bits por coordenada, formato 4.12. Si bit **V** es 1
	—		Z		
	Sólo si: **A=1** y **joint=255** y **N=1**. Muestras de todas las normales: $NN \times Samples$ (de **ske**). Primero todas las normales en orden de la primera muestra, y así de muestra en muestra.				
					Normal 10 bits por componente (V10), en formato 1.9
Para cada material de los **NM materiales** en el modelo					
	A2	TR	AR	flags	Opciones del material
					T bit 0: si este material tiene identificador de textura **T2** bit 1: si este material tiene segunda textura **TA** bit 2: si este material tiene la primera textura animada **AN** bit 3: si activa cíclicamente **AR** 5 bits (en byte): alpha de mezcla de capa de reflexión **TR** 5 bits (en byte): transparencia; 0 diáfano, 31 opaco **A2** 5 bits (en byte): alpha de mezcla de 2ª textura
	ambient		diffuse		Colores en formato 5 bits por componente
	emisive		specular		
					Si **T** activado hay textura, incluye identificador de hasta 12 caracteres de la textura
	Period		SIY	SIX	Si **TA=1** hay datos de textura animada, periodo en frames NDS (formato 12.4) y ancho **SIX** y alto **SIY** en subimágenes (8 bits)
					Si **T2** activado hay textura, incluye identificador de hasta 12 caracteres de la segunda textura

A.2. Directivas al conversor EDL

Con el fin de configurar ciertas opciones del modelo EDL en el momento de su conversión, se han establecido una serie de directivas que le dicen al conversor EDL qué parámetros activar o deshabilitar y en qué medida.

Estas directivas se establecen en la fase de modelado. El programa de modelado MilkShape 3D permite asociar a cada elemento del modelo (malla, material, articulación o modelo completo) un comentario, que posteriormente se almacena en el archivo Milkshape ASCII exportado. En estos comentarios es donde se insertan las directivas, las cuales el conversor EDL procesa, y aplica el atributo correspondiente en cada caso y en función del contexto.

La lista de directivas correspondiente a la última versión del formato EDL se muestra en la Tabla A.2

Tabla A.2: Directivas de la versión 3 del formato EDL.

Directivas			
Forma normal	**Abreviatura**	**Contexto**	**Significado**
#ALTO float #HEIGHT float		modelo	Alto del modelo (en eje y).
#ANCHO float #WIDTH float		modelo	Ancho del modelo (en eje x).
#FONDO float #DEPTH float		modelo	Fondo del modelo (en eje z).
#FACTORESCALA float #SCALEFACTOR float	#FE #SF	modelo	Factor de escala para el modelo.
#CAPAS reflec/tex2 tex2/reflec #LAYERS reflec/tex2 tex2/reflec		modelo	Orden de capas adicionales de rendering.
#SINNORMALES #NONORMALS	#SN #NN	modelo malla	No incluir las normales.
#NORMALES #NORMALS	#IN	modelo malla	Sí incluir las normales.
#SINCOORDTEX #NOTEXCOORD	#SCT #NTC	modelo malla	No incluir coordenadas de textura.
#COORDTEX #TEXCOORD	#ICT #ITC	modelo malla	Sí incluir coordenadas de textura.
#NS		malla	Malla no sólida (no hacer *backface culling*).
#SINTEXTURA #NOTEXTURE	#ST #NT	modelo material	No incluir identificador(es) de textura.
#TEXTURA #TEXTURE	#IT	modelo material	Incluir textura(s).

#V16		modelo malla	Coordenadas de vértices formato 16 bits (4.12 parte entera, fracción). Dos enteros de 32 bits por vértice.
#V10		modelo malla	Coordenadas de vértices en formato 10 bits (4.6 parte entera, fracción). Opción por defecto, 32 bits por vértice.
#CPV "fichero"		malla	Color por vértice tomado de fichero dado con su path, para esta malla.
#CPVD		malla	Color por vértice tomado de fichero con el nombre de la malla .cpv en la misma carpeta que MilShape ASCII.
#TEXTURAANIMADA #ANIMATEDTEXTURE	#TA #AT	material	Especificación de textura animada.
#DURACIONTA float #ATDURATION float	#DTA #ATD	material	Duración de textura animada.
#TAACTIVA #ATACTIVE		material	Textura animada activa al inicio.
#ALFA reflec/tex2 float #ALPHA reflec/tex2 float		material	Alfa para capa de rendering adicional.
#DURACION float #DURATION float	#D	articulación	Duración de animación completa.
#AP #PA		articulación	Animación Periódica.
#COSTUMBRES int #CUSTOMS int	#PAN	articulación	Costumbres de animación, frames finales.
#BBY		articulación	Billboard cilíndrico (en el eje Y).
#BBO #BB0		articulación	Billboard esférico.
#EP		articulación	Exportar Posición de Articulación.
#ER		articulación	Exportar Rotación de Articulación.
#MUESTREO float #SAMPLING float		modelo articulación	Periodo de Muestreo.

Apéndice B

Notas sobre animación por transformación y por muestreo de vértices

En este epígrafe se detallan algunos apuntes referentes al sistema de animación, en concreto a las diferencias en términos de eficiencia entre usar la animación por transformación y la animación por muestreo de vértices.

La animación por transformación es presumiblemente menos costosa en memoria, ya que con una transformación por muestra se animan todos los vértices asociados a la articulación, en lugar de tener que almacenar todos los vértices animados de la malla por muestra. Para que esta afirmación sea cierta, se tiene que verificar una ecuación en la que intervienen el número de articulaciones animadas por transformación, y el número de vértices y normales muestreados. Esta ecuación se halla estableciendo el número de unidades de memoria, en este caso 32 bits, que requieren cada uno de los métodos de animación. Para ello se establecen los parámetros siguientes:

p: unidad de almacenamiento (32 bits)

M: número de muestras de animación

Animación por transformación:

- t: memoria que ocupa una transformación (traslación + rotación); $t = 7p$ (3p de traslación y 4p de rotación)
- t_r: memoria que ocupa la traslación en reposo; $t_r = 3p$
- A: número de articulaciones

Animación por muestreo de vértices:

- v: memoria que ocupa un vértice; $v = p$ (en el caso que se usen V10, en el que cada componente ocupa 10 bits, sino $v = 2p$)
- n: memoria que ocupa una normal; $n = p$
- V: número de vértices
- N: número de normales

Tratando primero el método por transformación, el cálculo de la memoria ocupada por

las transformaciones Pt sería

$$Pt = M \times A \times t + A \times t_r, \tag{B.1}$$

que factorizando queda

$$P_t = A \times (7 \times M \times p + 3 \times p). \tag{B.2}$$

En este caso, se despreciará en la ecuación anterior tr, ya que el número de muestras suele ser superior a 25 o 30, resultando 175 frente a 3, en el caso de tener 25 muestras por animación (incluso siendo éste un número bajo). Por lo tanto la ecuación quedaría:

$$P_t = 7 \times A \times M \times p. \tag{B.3}$$

En el caso de la animación por muestreo de vértices, el cálculo sería

$$P_v = M \times (V \times v + N \times n). \tag{B.4}$$

Simplificando y sustituyendo por p (en el caso de vértices V10):

$$P_v = M \times p \times (V + N). \tag{B.5}$$

Por lo tanto, la afirmación planteada es cierta si se cumple

$$Pt < Pv, \tag{B.6}$$

es decir,

$$7 \times A < V + N. \tag{B.7}$$

Tomemos algunas referencias: el formato EDL permite hasta 1024 vértices, 1024 normales y 254 articulaciones por modelo. En el caso más extremo en el que se alcancen estos valores límite, $7 \times A$ sería 1778 (en el caso de animación por transformación) y $V + N$ sería 2048 en el caso de utilizar V10 y $2 \times V + N$ sería 3072 en el caso de utilizar V16 (en el caso de animación por muestreo de vértices). En cada caso se aumenta en 15 % y 72 % la memoria empleada con animación por transformación frente a la animación por muestreo de vértices. En este caso compensaría por lo tanto emplear la animación por transformación siempre que se pudiera, considerando únicamente el consumo de memoria como parámetro de decisión.

En un caso más práctico: un modelo de referencia con una densidad poligonal media rondaría los 150 polígonos, que con el cálculo rápido de 3 vértices por polígono quedarían 300 vértices. Se tomará el número de normales igual al de vértices. Para una animación de un personaje bípedo simple, harían falta como unas 12 articulaciones tirando por lo alto, ya que algunas son prescindibles: cuello, hombros, codos, muñecas, cintura, cabezas del fémur, rodillas y tobillos. Por lo tanto, en este caso, para la animación por transformación, $7 \times A$ sería 84, mientras que para la animación por muestreo de vértices, si se emplea

V10, $V + N$ sería 600 y si se emplea V16, $2 \times V + N$ sería 900. En este caso, la diferencia es de 14 % y 71 % (V10 y V16 respectivamente) entre emplear un método u otro.

Por lo tanto, en un entorno práctico, la animación por transformación es un método favorable para ahorrar memoria .

Sin embargo, la animación por muestreo de vértices presenta una ventaja frente al método por transformación: el ahorro de cálculos en el renderizado. Cuando se renderiza un modelo, a cada ciclo se le pasan al hardware los vértices del modelo, éste se mueva o no. En el caso de animarlo por muestreo de vértices, se le pasa el mismo número de vértices pero con coordenadas diferentes. Por lo tanto el número de cálculos que debe realizar el chip gráfico es el mismo. Sin embargo, en el caso de la animación por transformación, por cada traslación y cada rotación de cada una de las articulaciones hay que realizar una multiplicación matricial. Y esto para cada muestra de la animación. Por lo tanto, el número de cálculos adicionales es considerablemente más alto, enlenteciendo el proceso global.

De esta forma, hay un cierto equilibrio entre ambos métodos de animación, para los que hay que controlar, por una parte el número de articulaciones que animar, y por otra el número de vértices y normales que muestrear, para que el renderizado no se vea enlentecido en exceso y no consumir demasiada memoria que se pudría utilizar para almacenar más recursos.

Referencias

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Autodesk 3D Studio Max. http://www.autodesk.com/products/autodesk-3ds-max/overview.

[3] Autodesk Maya. http://www.autodesk.com/products/autodesk-maya/overview.

[4] Blender. http://www.blender.org/.

[5] Blender NDS Exporter. https://github.com/kiniou/blender-nds-exporter/wiki.

[6] Wayne E. Carlson. Cgi historical timeline. Web site: http://excelsior.biosci.ohio-state.edu/~carlson/history/timeline.html, 2004.

[7] DeSmuME. http://www.desmume.com/.

[8] DevkitPro. http://devkitpro.org/.

[9] DirectX. http://directx.es/.

[10] Gbatek. Información técnica de la nintendo ds y gba. http://nocash.emubase.de/gbatek.htm.

[11] Grit. http://www.coranac.com/projects/grit/.

[12] iDeas. http://ciacin.site90.com/ideas.php.

[13] Libnds. http://libnds.devkitpro.org/.

[14] MilkShape 3D. http://chumbalum.swissquake.ch/index.html.

[15] NFlib. http://www.nightfoxandco.com/index.php/20120318/nflib-version-20120318/.

[16] Antonio Niño (NightFox). Tutorial de instalación del entorno de desarrollo hombrew de la Nintendo DS. http://www.nightfoxandco.com/index.php/main-es/programacion/primerospasos/.

[17] Nitro Engine. http://antoniond.drunkencoders.com/nitroengine.html.

[18] NO$GBA. http://www.nogba.com/.

[19] OpenGL. http://www.opengl.org/.

[20] M. M. d. C. V. Patiño. Programación de gráficas computacionales. Web site: http://educommons.anahuac.mx:8080/eduCommons/computacion-y-sistemas/programacion-de-graficas-computacionales, Febrero 2010. RetrievedFebruary

10, 2013, fromOpenCourseWare de la Universidad Anáhuac México Norte.

[21] Wikipedia. Animación por esqueleto. http://en.wikipedia.org/wiki/Skeletal_animation.

[22] Wikipedia. Nurbs. http://es.wikipedia.org/wiki/NURBS.

[23] Wikipedia. Patrones de Moiré. http://en.wikipedia.org/wiki/Moire_pattern.